ModelSim® SE User's Manual

Software Version 6.2a

June 2006

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: www.mentor.com/supportnet
Send Feedback on Documentation: www.mentor.com/supportnet/documentation/reply_form.cfm

Table of Contents

Chapter 1

Chapter 2

Chapter 3

Chapter 6

Chapter 7

Chapter 9

Chapter 10

Chapter 11

Chapter 12

Chapter 13

Chapter 14

Chapter 19

Chapter 20

Chapter 21

Appendix B
Location Mapping

Appendix C
Error and Warning Messages

Appendix D
Verilog PLI/VPI/DPI

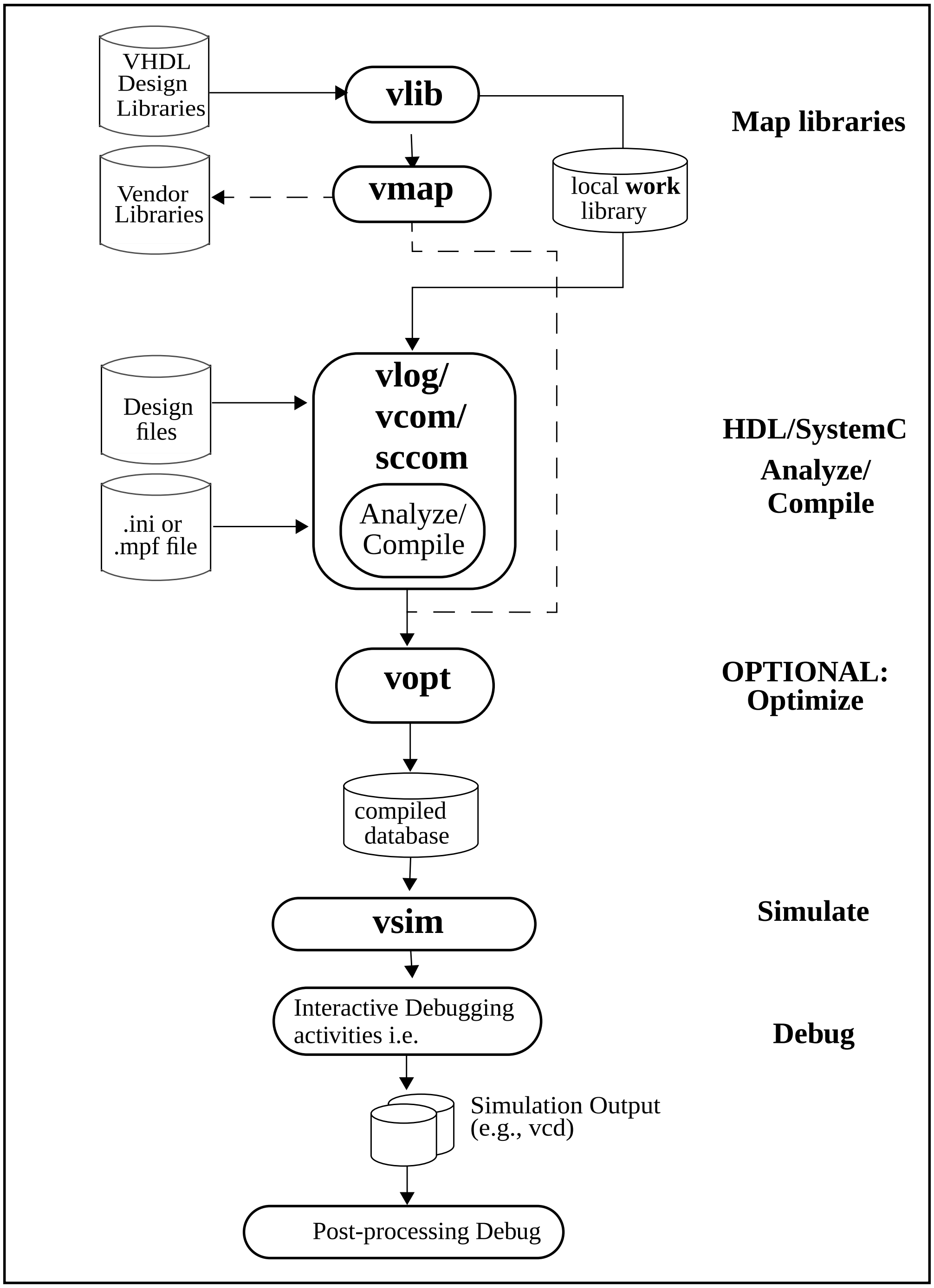
List of Examples

Chapter 1
Introduction

This documentation was written for UNIX and Microsoft Windows users. Not all versions of ModelSim are supported on all platforms. Contact your Mentor Graphics sales representative for details.

Tool Structure and Flow

The diagram below illustrates the structure of the ModelSim tool, and the flow of that tool as it is used to verify a design.



Simulation Task Overview

The following table provides a reference for the tasks required for compiling, optimizing, loading, and simulating a design in ModelSim.

Table 1-1.

Task	**Example Command Line Entry**	**GUI Menu Pull-down**	**GUI Icons**
Step 1: Map libraries	**vlib** <library_name> **vmap** work <library_name>	1. **File > New > Project** 2. Enter library name 3. Add design files to project	N/A
Step 2: Compile the design	**vlog** file1.v file2.v ... (Verilog) **vcom** file1.vhd file2.vhd ... (VHDL) **sccom** <top> (SystemC) **sccom -link** <top>	**Compile > Compile** or **Compile > Compile All**	**Compile** or **Compile All** icons:
Step 3: Optimize the design (OPTIONAL)	**voptflow = 1** in modelsim.ini file OR **vopt** <top> -o <opt_name>	To enable optimization: 1. **Simulate > Start Simulation** 2. **Enable Optimization** button To set optimization options: 1. **Simulate > Design Optimization** 2. Set desired optimizations	N/A
Step 4: Load the design into the simulator	**vsim** <top> or **vsim** <opt_name>	1. **Simulate > Start Simulation** 2. Click on top design module or optimized design unit name 3. Click OK This action loads the design for simulation.	**Simulate** icon:
Step 5: Run the simulation	run step	**Simulate > Run**	**Run**, or **Run continue**, or **Run -all** icons:

Table 1-1.

Task	Example Command Line Entry	GUI Menu Pull-down	GUI Icons
Step 6: Debug the design Note: Design optimization in step 3 limits debugging visibility	Common debugging commands: bp describe drivers examine force log show	**N/A**	N/A

Basic Steps for Simulation

This section provides further detail related to each step in the process of simulating your design using ModelSim.

Step 1 — Collecting Files and Mapping Libraries

Files needed to run ModelSim on your design:

- design files (VHDL, Verilog, and/or SystemC), including stimulus for the design
- libraries, both working and resource
- modelsim.ini (automatically created by the library mapping command

Providing Stimulus to the Design

You can provide stimulus to your design in several ways:

- Language based testbench
- Tcl-based ModelSim interactive command, force
- VCD files / commands

  See Creating a VCD File and Using Extended VCD as Stimulus
- 3rd party testbench generation tools

What is a Library?

A library is a location where data to be used for simulation is stored. Libraries are ModelSim’s way of managing the creation of data before it is needed for use in simulation. It also serves as a way to streamline simulation invocation. Instead of compiling all design data each and every time you simulate, ModelSim uses binary pre-compiled data from these libraries. So, if you make a changes to a single Verilog module, only that module is recompiled, rather than all modules in the design.

Working and Resource Libraries

Design libraries can be used in two ways: 1) as a local working library that contains the compiled version of your design; 2) as a resource library. The contents of your working library will change as you update your design and recompile. A resource library is typically unchanging, and serves as a parts source for your design. Examples of resource libraries might be: shared information within your group, vendor libraries, packages, or previously compiled elements of your own working design. You can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor).

For more information on resource libraries and working libraries, see Working Library Versus Resource Libraries, Managing Library Contents, Working with Design Libraries, and Specifying the Resource Libraries.

Creating the Logical Library (vlib)

Before you can compile your source files, you must create a library in which to store the compilation results. You can create the logical library using the GUI, using **File > New > Library** (see Creating a Library), or you can use the vlib command. For example, the command:

```
vlib work
```

creates a library named **work**. By default, compilation results are stored in the **work** library.

Mapping the Logical Work to the Physical Work Directory (vmap)

VHDL uses logical library names that can be mapped to ModelSim library directories. If libraries are not mapped properly, and you invoke your simulation, necessary components will not be loaded and simulation will fail. Similarly, compilation can also depend on proper library mapping.

By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI (Library Mappings with the GUI, a command (Library Mapping from the Command Line), or a project (Getting Started with Projects to assign a logical name to a design library.

The format for command line entry is:

```
vmap <logical_name> <directory_pathname>
```

This command sets the mapping between a logical library name and a directory.

Step 2 — Compiling the Design (vlog, vcom, sccom)

Designs are compiled with one of the three language compilers.

Compiling Verilog (vlog)

ModelSim's compiler for the Verilog modules in your design is vlog. Verilog files may be compiled in any order, as they are not order dependent. See Compiling Verilog Files for details.

Compiling VHDL (vcom)

ModelSim's compiler for VHDL design units is vcom. VHDL files must be compiled according to the design requirements of the design. Projects may assist you in determining the compile order: for more information, see Auto-Generating Compile Order**.** See Compiling VHDL Files for details. on VHDL compilation.

Compiling SystemC (sccom)

ModelSim's compiler for SystemC design units is sccom, and is used only if you have SystemC components in your design. See Compiling SystemC Files for details.

Step 3 — Loading the Design for Simulation

vsim topLevelModule

Your design is ready for simulation after it has been compiled. You may then invoke vsim with the names of the top-level modules (many designs contain only one top-level module). For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references.

You can optionally optimize the design with vopt. For more information on optimization, see Optimizing Designs with vopt.

Using SDF

You can incorporate actual delay values to the simulation by applying SDF back-annotation files to the design. For more information on how SDF is used in the design, see Specifying SDF Files for Simulation.

Step 4 — Simulating the Design

Once the design has been successfully loaded, the simulation time is set to zero, and you must enter a **run** command to begin simulation. For more information, see Verilog and SystemVerilog Simulation, SystemC Simulation, and VHDL Simulation.

The basic simulator commands are:

- add wave
- force
- bp
- run
- step

Step 5 — Debugging the Design

Numerous tools and windows useful in debugging your design are available from the ModelSim GUI. In addition, several basic simulation commands are available from the command line to assist you in debugging your design:

- describe
- drivers
- examine
- force
- log
- checkpoint
- restore
- show

Modes of Operation

Many users run ModelSim interactively–pushing buttons and/or pulling down menus in a series of windows in the GUI (graphical user interface). But there are really three modes of ModelSim operation, the characteristics of which are outlined in the following table.:

Table 1-2.

ModelSim use mode	Characteristics	How ModelSim is invoked
GUI	interactive; has graphical windows, push-buttons, menus, and a command line in the transcript. Default mode.	via a desktop icon or from the OS command shell prompt. Example: `OS> vsim`
Command-line	interactive command line; no GUI.	with **-c** argument at the OS command prompt. Example: `OS> vsim -c`
Batch	non-interactive batch script; no windows or interactive command line.	at OS command shell prompt using "here document" technique or redirection of standard input. Example: `C:\ vsim vfiles.v <infile >outfile`

The ModelSim User’s Manual focuses primarily on the GUI mode of operation. However, this section provides an introduction to the Command-line and Batch modes.

Command Line Mode

In command line mode ModelSim executes any startup command specified by the Startup variable in the *modelsim.ini* file. If vsim is invoked with the **-do "command_string"** option, a DO file (macro) is called. A DO file executed in this manner will override any startup command in the *modelsim.ini* file.

During simulation a transcript file is created containing any messages to stdout. A transcript file created in command line mode may be used as a DO file if you invoke the transcript **on** command after the design loads (see the example below). The transcript **on** command writes all of the commands you invoke to the transcript file. For example, the following series of commands results in a transcript file that can be used for command input if *top* is re-simulated (remove the **quit -f** command from the transcript file if you want to remain in the simulator).

```
vsim -c top
```

library and design loading messages… then execute:

```
transcript on
force clk 1 50, 0 100 -repeat 100
run 500
run @5000
quit -f
```

Rename transcript files that you intend to use as DO files. They will be overwritten the next time you run **vsim** if you don’t rename them. Also, simulator messages are already commented out, but any messages generated from your design (and subsequently written to the transcript file) will cause the simulator to pause. A transcript file that contains only valid simulator commands will work fine; comment out anything else with a "#".

Stand-alone tools pick up project settings in command line mode if they are invoked in the project's root directory. If invoked outside the project directory, stand-alone tools pick up project settings only if you set the **MODELSIM** environment variable to the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

Batch Mode

Batch mode is an operational mode that provides neither an interactive command line nor interactive windows. In a Windows environment, **vsim** is run from a Windows command prompt and standard input and output are redirected from and to files.

In a UNIX environment, **vsim** can be invoked in batch mode by redirecting standard input using the “here-document” technique.

Here is an example of the "here-document" technique:

```
vsim top <<!
log -r *
run 100
do test.do
quit -f
!
```

Here is an example of a batch mode simulation using redirection of std input and output:

```
vsim counter < yourfile > outfile
```

where "yourfile" is a script containing various ModelSim commands.

You can use the CTRL-C keyboard interrupt to break batch simulation in UNIX and Windows environments.

Graphic Interface Overview

While your operating system interface provides the window-management frame, ModelSim controls all internal-window features including menus, buttons, and scroll bars. The resulting simulator interface remains consistent within these operating systems:

- SPARCstation with OpenWindows, OSF/Motif, or CDE

- IBM RISC System/6000 with OSF/Motif
- Hewlett-Packard HP 9000 Series 700 with HP VUE, OSF/Motif, or CDE
- Redhat or SuSE Linux with KDE or GNOME
- Microsoft Windows 98/Me/NT/2000/XP

Because ModelSim's graphic interface is based on Tcl/TK, you also have the tools to build your own simulation environment. Preference variables and configuration commands (see Control Variables Located in INI Files for details) give you control over the use and placement of windows, menus, menu options, and buttons. See Tcl and Macros (DO Files) for more information on Tcl.

Standards Supported

ModelSim VHDL implements the VHDL language as defined by IEEE Standards 1076-1987, 1076-1993, and 1076-2002. ModelSim also supports the 1164-1993 *Standard Multivalue Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with ModelSim will be compatible with any other VHDL system that is compliant with the 1076 specs.

ModelSim Verilog implements the Verilog language as defined by the IEEE Std 1364-1995 and 1364-2005. ModelSim Verilog also supports a partial implementation of SystemVerilog P1800-2005 (see */<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details). Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim users.

In addition, all products support SDF 1.0 through 4.0 (except the NETDELAY statement), VITAL 2.2b, VITAL'95 – IEEE 1076.4-1995, and VITAL 2000 – IEEE 1076.4-2000.

ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.1 reference simulator.

Assumptions

We assume that you are familiar with the use of your operating system and its graphical interface.

We also assume that you have a working knowledge of the design languages. Although ModelSim is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Finally, we assume that you have worked the appropriate lessons in the *ModelSim Tutorial* and are familiar with the basic functionality of ModelSim. The *ModelSim Tutorial* is available from the ModelSim **Help** menu.

The *ModelSim Tutorial* is also available from the Support page of our web site:

`www.model.com`

Sections In This Document

In addition to this introduction, you will find the following major sections in this document:

Chapter 4, Projects — This chapter discusses ModelSim "projects", a container for design files and their associated simulation properties.

Chapter 5, Design Libraries — To simulate an HDL design using ModelSim, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

Chapter 6, VHDL Simulation — This chapter is an overview of compilation and simulation for VHDL within the ModelSim environment.

Chapter 7, Verilog and SystemVerilog Simulation — This chapter is an overview of compilation and simulation for Verilog and SystemVerilog within the ModelSim environment.

Chapter 8, SystemC Simulation — This chapter is an overview of preparation, compilation, and simulation for SystemC within the ModelSim environment.

Chapter 9, Mixed-Language Simulation — This chapter outlines data mapping and the criteria established to instantiate design units between languages.

Chapter 11, WLF Files (Datasets) and Virtuals — This chapter describes datasets and virtuals - both methods for viewing and organizing simulation data in ModelSim.

Chapter 12, Waveform Analysis — This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

Chapter 13, Tracing Signals with the Dataflow Window — This chapter describes how to trace signals and assess causality using the ModelSim Dataflow window.

Chapter 14, Measuring Coverage — This chapter describes the Code Coverage feature. Code Coverage gives you graphical and report file feedback on how the source code is being executed.

Chapter 16, C Debug — This chapter describes C Debug, a graphic interface to the **gdb** debugger that can be used to debug FLI/PLI/VPI/SystemC C/C++ source code.

Chapter 17, Profiling Performance and Memory Use — This chapter describes how the ModelSim Performance Analyzer is used to easily identify areas in your simulation where performance can be improved.

Chapter 18, Signal Spy — This chapter describes Signal Spy, a set of VHDL procedures and Verilog system tasks that let you monitor, drive, force, or release a design object from anywhere in the hierarchy of a VHDL or mixed design.

Chapter 19, Monitoring Simulations with JobSpy — This chapter describes JobSpy^{TM}, a tool for monitoring and controlling batch simulations and simulation farms.

Chapter 20, Generating Stimulus with Waveform Editor — This chapter describes how to perform waveform analysis with the ModelSim Wave and List windows.

Chapter 21, Standard Delay Format (SDF) Timing Annotation — This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Chapter 22, Value Change Dump (VCD) Files — This chapter explains Model Technology's Verilog VCD implementation for ModelSim. The VCD usage is extended to include VHDL designs.

Chapter 23, Tcl and Macros (DO Files) — This chapter provides an overview of Tcl (tool command language) as used with ModelSim.

Appendix A, Simulator Variables — This appendix describes environment, system, and preference variables used in ModelSim.

Appendix C, Error and Warning Messages — This appendix describes ModelSim error and warning messages.

Appendix D, Verilog PLI/VPI/DPI — This appendix describes the ModelSim implementation of the Verilog PLI and VPI.

Appendix E, Command and Keyboard Shortcuts — This appendix describes ModelSim keyboard and mouse shortcuts.

Appendix G, System Initialization — This appendix describes what happens during ModelSim startup.

Appendix I, Logic Modeling SmartModels — This appendix describes the use of the SmartModel Library and SmartModel Windows with ModelSim.

Appendix H, Logic Modeling Hardware Models — This appendix describes the use of the Logic Modeling Hardware Modeler with ModelSim.

What is an "Object"

Because ModelSim works with so many languages (SystemC, Verilog, VHDL, SystemVerilog,), an “object” refers to any valid design element in those languages. The word "object" is used whenever a specific language reference is not needed. Depending on the context, “object” can refer to any of the following:

Table 1-3.

Language	An object can be
VHDL	block statement, component instantiation, constant, generate statement, generic, package, signal, alias, or variable
Verilog	function, module instantiation, named fork, named begin, net, task, register, or variable
SystemVerilog	In addition to those listed above for Verilog: class, package, program, interface, array, directive, property, or sequence
SystemC	module, channel, port, variable, or aggregate
PSL	property, sequence, directive, or endpoint

Text Conventions

Text conventions used in this manual include:

Table 1-4.

Text Type	Description
italic text	provides emphasis and sets off filenames, pathnames, and design unit names
bold text	indicates commands, command options, menu choices, package and library logical names, as well as variables, dialog box selections, and language keywords
`monospace type`	monospace type is used for program and command examples
The right angle (>)	is used to connect menu choices when traversing menus as in: **File** > **Quit**
path separators	examples will show either UNIX or Windows path separators - use separators appropriate for your operating system when trying the examples
UPPER CASE	denotes file types used by ModelSim (e.g., DO, WLF, INI, MPF, PDF, etc.)

Installation Directory Pathnames

When referring to installation paths, this manual uses "modeltech" as a generic representation of the installation directory for all versions of ModelSim. The actual installation directory on your system may contain version information.

Where to Find Our Documentation

ModelSim documentation is available from our website at

`www.model.com/support`

or from the tool in the following formats and locations:

Table 1-5.

Document	Format	How to get it
Installation & Licensing Guide	PDF	Help > PDF Bookcase
	HTML	**Help > Help & Manuals** displays the InfoHub Select the **Release Management** tab
Quick Guide (command and feature quick-reference)	PDF	Help > PDF Bookcase
Tutorial	PDF	Help > PDF Bookcase
	HTML	**Help > Help & Manuals** displays the InfoHub Select the **Support & Training** tab
User's Manual	PDF	Help > PDF Bookcase
	HTML	**Help > Help & Manuals** displays the InfoHub Select the **Help & Manuals** tab
Reference Manual	PDF	Help > PDF Bookcase
	HTML	**Help > Help & Manuals** displays the InfoHub Select the **Help & Manuals** tab
Foreign Language Interface Manual	PDF	Help > PDF Bookcase
	HTML	**Help > Help & Manuals** displays the InfoHub Select the **Help & Manuals** tab
Std_DevelopersKit User's Manual	PDF	www.model.com/support/documentation/BOOK/sdk_um.pdf The Standard Developer's Kit is for use with Mentor Graphics QuickHDL.
Command Help	ASCII	type **help [command name]** at the prompt in the Transcript pane

Table 1-5.

Document	Format	How to get it
Error message help	ASCII	type **verror <msgNum>** at the Transcript or shell prompt
Tcl Man Pages (Tcl manual)	HTML	select **Help > Tcl Man Pages**, or find *contents.htm* in *\modeltech\docs\tcl_help_html*
Technotes	HTML	available from the support site

Download a Free PDF Reader With Search

ModelSim PDF documentation requires an Adobe Acrobat Reader for viewing. The Reader is available without cost from Adobe at

`www.adobe.com.`

Acrobat Reader allows you to take advantage of the index file supplied with our documentation; the index makes searching for keywords much faster.

Technical Support and Updates

Support

Online and email technical support options, maintenance renewal, and links to international support contacts:

`www.model.com/support/default.asp`

Updates

Access to the most current version of ModelSim:

`www.model.com/downloads/default.asp`

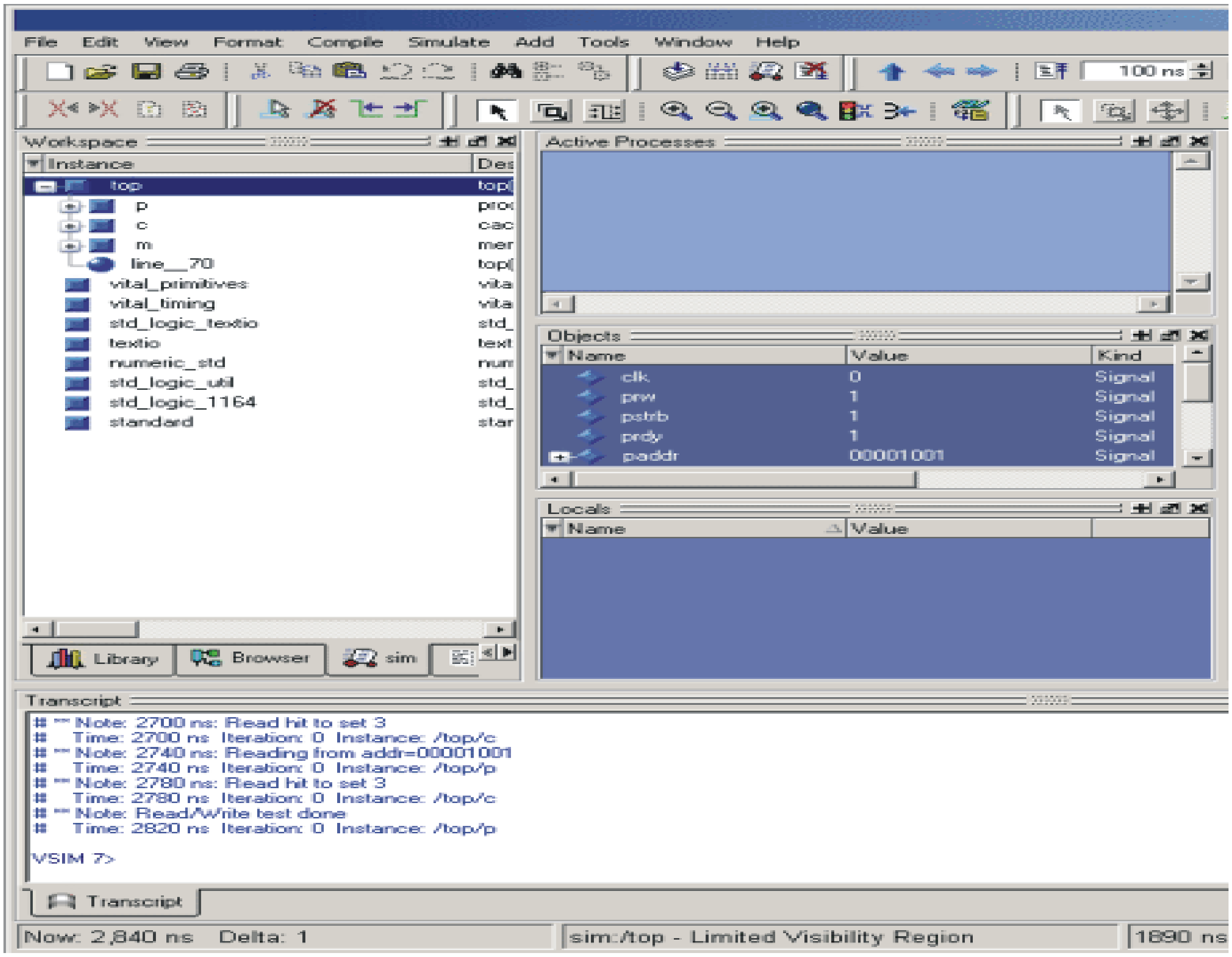
Latest Version Email

Place your name on our list for email notification of news and updates:

`ww.model.com/products/informant.asp`

Chapter 2
Simulator Windows

ModelSim’s graphical user interface (GUI) consists of various windows that give access to parts of your design and numerous debugging tools. Some of the windows display as panes within the ModelSim Main window, some display as windows in the Multiple Document Interface (MDI) frame, and some display as standalone windows.



Name	Value	Kind
clk	0	Signal
prw	1	Signal
pstrb	1	Signal
prdy	1	Signal
paddr	00001001	Signal

```
# ** Note: 2700 ns: Read hit to set 3
#    Time: 2700 ns  Iteration: 0  Instance: /top/c
# ** Note: 2740 ns: Reading from addr=00001001
#    Time: 2740 ns  Iteration: 0  Instance: /top/p
# ** Note: 2780 ns: Read hit to set 3
#    Time: 2780 ns  Iteration: 0  Instance: /top/c
# ** Note: Read/Write test done
#    Time: 2820 ns  Iteration: 0  Instance: /top/p

VSIM 7>
```

Now: 2,840 ns   Delta: 1   sim:/top - Limited Visibility Region   1890 ns

The following table summarizes all of the available windows and panes.

Table 2-1.

Window/pane name	Description	More details
Main	central GUI access point	Main Window
Active Processes	displays all processes that are scheduled to run during the current simulation cycle	Active Processes Pane
Code coverage	a collection of panes that display code coverage data	Code Coverage Panes
Dataflow	displays "physical" connectivity and lets you trace events (causality)	Dataflow Window
List	shows waveform data in a tabular format	List Window
Locals	displays data objects that are immediately visible at the current PC of the selected process	Locals Pane
Memory	a Workspace tab and MDI windows that show memories and their contents	Memory Panes
Watch	displays signal or variable values at the current simulation time	Watch Pane
Objects	displays all declared data objects in the current scope	Objects Pane
Profile	two panes that display performance and memory profiling data	Profile Panes
Source	a text editor for viewing and editing HDL, SystemC, DO, etc. files	Source Window
Transcript	keeps a running history of commands and messages and provides a command-line interface	Transcript
Wave	displays waveforms	Wave Window
Workspace	provides easy access to projects, libraries, compiled design units, memories, etc.	Workspace

The windows and panes are customizable in that you can position and size them as you see fit, and ModelSim will remember your settings upon subsequent invocations. See Navigating the Graphic User Interface for more details.

Design Object Icons and Their Meaning

The color and shape of icons convey information about the language and type of a design object. Here is a list of icon colors and the languages they indicate:

icon color	**language**
light blue	Verilog or SystemVerilog
dark blue	VHDL
green	SystemC
orange	virtual object

Here is a list of icon shapes and the design object types they indicate:

Table 2-2.

icon shape	**example**	**design object type**
square	■	any scope (VHDL block, Verilog named block, SC module, class, interface, task, function, etc.)
circle	●	process
diamond	◆	valued object (signals, nets, registers, SystemC channel, etc.)
caution sign	⚠	comparison object
diamond with red dot	◆	an editable waveform created with the waveform editor

Setting Fonts

You may need to adjust font settings to accommodate the aspect ratios of wide screen and double screen displays or to handle launching ModelSim from an X-session.

Font Scaling

To change font scaling, select **Tools** > **Options** > **Adjust Font Scaling**. You’ll need a ruler to complete the instructions in the lower right corner of the dialog. When you have entered the pixel and inches information, click OK to close the dialog. Then, restart ModelSim to see the change. This is a one time setting; you shouldn't have to set it again unless you change display resolution or the hardware (monitor or video card). The font scaling applies to Windows and UNIX operating systems. On UNIX systems, the font scaling is stored based on the $DISPLAY environment variable.

Controlling Fonts in an X-session

When executed via an X-session (e.g., Exceed, VNC), ModelSim uses font definitions from the .Xdefaults file. To ensure that the fonts look correct, create a .Xdefaults file with the following lines:

```
vsim*Font: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*SystemFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*StandardFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
vsim*MenuFont: -adobe-courier-medium-r-normal--*-120-*-*-*-*-*
```

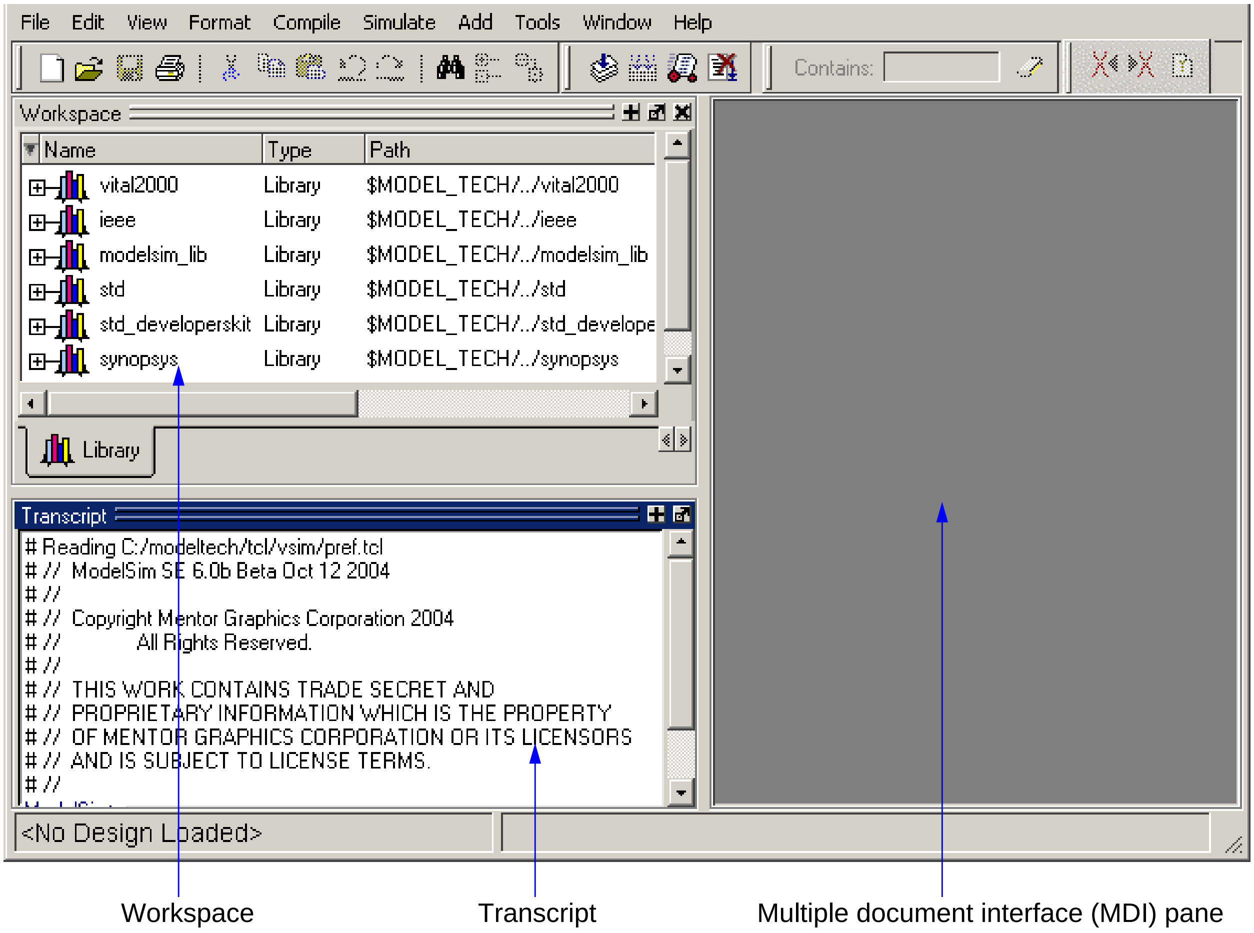
Alternatively, you can choose a different font. Use the program "xlsfonts" to identify which fonts are available on your system.

Also, the following command can be used to update the X resources if you make changes to the .Xdefaults and wish to use those changes on a UNIX machine:

```
xrdb -merge .Xdefaults
```

Main Window

The primary access point in the ModelSim GUI is called the Main window. Here is what the Main window looks like the very first time you start the tool:


File Edit View Format Compile Simulate Add Tools Window Help
Contains:
Workspace
Library
Transcript
Workspace
Transcript
Multiple document interface (MDI) pane

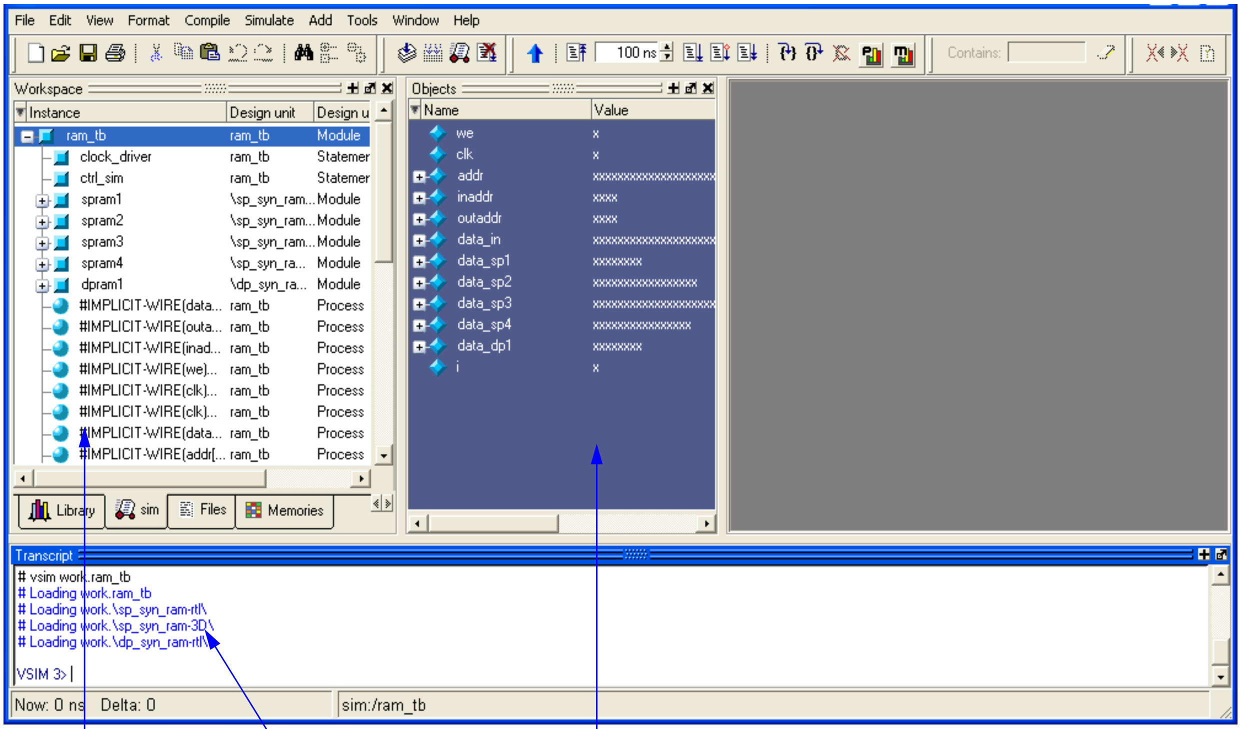
Name	Type	Path
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_develope
synopsys	Library	$MODEL_TECH/../synopsys

```
# Reading C:/modeltech/tcl/vsim/pref.tcl
# //  ModelSim SE 6.0b Beta Oct 12 2004
# //
# //  Copyright Mentor Graphics Corporation 2004
# //              All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
```

<No Design Loaded>

The Main window provides convenient access to design libraries and objects, source files, debugging commands, simulation status messages, etc.

When you load a design, or bring up debugging tools, ModelSim adds additional panes or opens new windows. For example, here is the Main window after loading a simple design.



Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns   Delta: 0   sim:/ram_tb

Workspace tabs organize design elements in a hierarchical tree structure

The Transcript pane reports status and provides a command-line interface

The Objects pane displays data objects in the current scope

Notice some of the elements that appear:

- Workspace tabs organize and display design objects in a hierarchical tree format
- The Transcript pane tracks command history and messages and provides a command-line interface where you can enter ModelSim commands
- The Objects pane displays design objects such as signals, nets, generics, etc. in the current design scope

Workspace

The Workspace provides convenient access to projects, libraries, design files, compiled design units, simulation/dataset structures, and Waveform Comparison objects. It can be hidden or displayed by selecting **View > Windows > Workspace** (Main window).

The Workspace can display the types of tabs listed below.

- **Project tab** — Shows all files that are included in the open project. Refer to Projects for details.
- **Library tab** — Shows design libraries and compiled design units. To update the current view of the library, select a library, and then Right click > Update. See Managing Library Contents for details on library management.
- **Structure tabs** —Shows a hierarchical view of the active simulation and any open datasets. There is one tab for the current simulation (named "sim") and one tab for each open dataset. See Viewing Dataset Structure for details.

  An entry is created by each object within the design. When you select a region in a structure tab, it becomes the *current region* and is highlighted. The Source Window and Objects Pane change dynamically to reflect the information for the current region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

  Also, when you select a region in the structure pane, the Active Processes Pane is updated. The Active Processes window will in turn update the Locals Pane.

  Objects can be dragged from the structure tabs to the Dataflow, List and Wave windows.

  The structure tabs will display code coverage information (see Viewing Coverage Data in the Main Window).

  You can toggle the display of processes by clicking in a Structure tab and selecting **View** > **Filter** > **Processes**.

  You can also control implicit wire processes using a preference variable. By default Structure tabs display implicit wire processes. To hide implicit wire processes permanently, set PrefStructure(HideImplicitWires) to 1 (select **Tools** > **Edit Preferences**, By Name tab, and expand the Structure object).
- **Files tab** — Shows the source files for the loaded design.

  You can disable the display of this tab by setting the PrefMain(ShowFilePane) preference variable to 0. See Simulator GUI Preferences for information on setting preference variables.

  The file tab will display code coverage information (see Viewing Coverage Data in the Main Window).
- **Memories tab** — Shows a hierarchical list of all memories in the design. To display this tab, select **View** > **Windows** > **Memory**. When you select a memory on the tab, a memory contents page opens in the MDI frame. See Memory Panes.
- Browser tab — Shows an expandable list of symbols from any files you have open in the MDI. You can display this tab by selecting **View** > **Windows** > **Code Browser**.

- Double-click a symbol name — changes the focus of the file in the MDI so that you can view the declaration.
- Right-click a symbol name and select References — opens a dialog box listing all file names and line numbers that refer to this symbol.

- **Compare tab** — Shows comparison objects that were created by doing a waveform comparison. See Waveform Analysis for details.

Transcript

The Transcript portion of the Main window maintains a running history of commands that are invoked and messages that occur as you work with ModelSim. When a simulation is running, the Transcript displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface.

You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also use arrow keys to recall previous commands, or copy and paste using the mouse within the window (see Main and Source Window Mouse and Keyboard Shortcuts for details).

Saving the Transcript File

Variable settings determine the filename used for saving the transcript. If either **PrefMain(file)** in the *.modelsim* file or **TranscriptFile** in the *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the **TranscriptFile** variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, click in the Transcript pane and then select **File** > **Save As**, or **File** > **Save**. The initial save must be made with the **Save As** selection, which stores the filename in the Tcl variable **PrefMain(saveFile)**. Subsequent saves can be made with the **Save** selection. Since no automatic saves are performed for this file, it is written only when you invoke a **Save** command. The file is written to the specified directory and records the contents of the transcript at the time of the save.

Using the Saved Transcript as a Macro (DO file)

Saved transcript files can be used as macros (DO files). Refer to the do command for more information.

Changing the Number of Lines Saved in the Transcript Window

By default, the Transcript window retains the last 5000 lines of output from the transcript. You can change this default by altering the saveLines preference variable. Setting this variable to 0 instructs the tool to retain all lines of the transcript. See Simulator GUI Preferences for details on setting preference variables.

Disabling Creation of the Transcript File

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Automatic Command Help

When you start typing a command at the Transcript prompt, a dropdown box appears which lists the available commands matching what has been typed so far. You may use the Up and Down arrow keys or the mouse to select the desired command. When a unique command has been entered, the command usage is presented in the drop down box.

You can disable this feature by selecting **Help > Command Completion** or by setting the *PrefMain(EnableCommandHelp)* preference variable to 0. See Simulator GUI Preferences for details on setting preference variables.

Message Viewer

The Message Viewer tab, found in the Transcript pane, allows you to easily access, organize, and analyze any Note, Warning, Error or other elaboration and runtime messages written to the transcript during the simulation run.

By default, the tool writes transcripted messages to both the transcript and the WLF file. By writing to the WLF file, the Message Viewer tab is able to organize the messages for your analysis.

Controlling the Message Viewer Data

- **Command Line** — The -msgmode argument to vsim controls where the simulator outputs the messages.

  ```
  vsim -msgmode {both | tran | wlf}
  ```

  where:

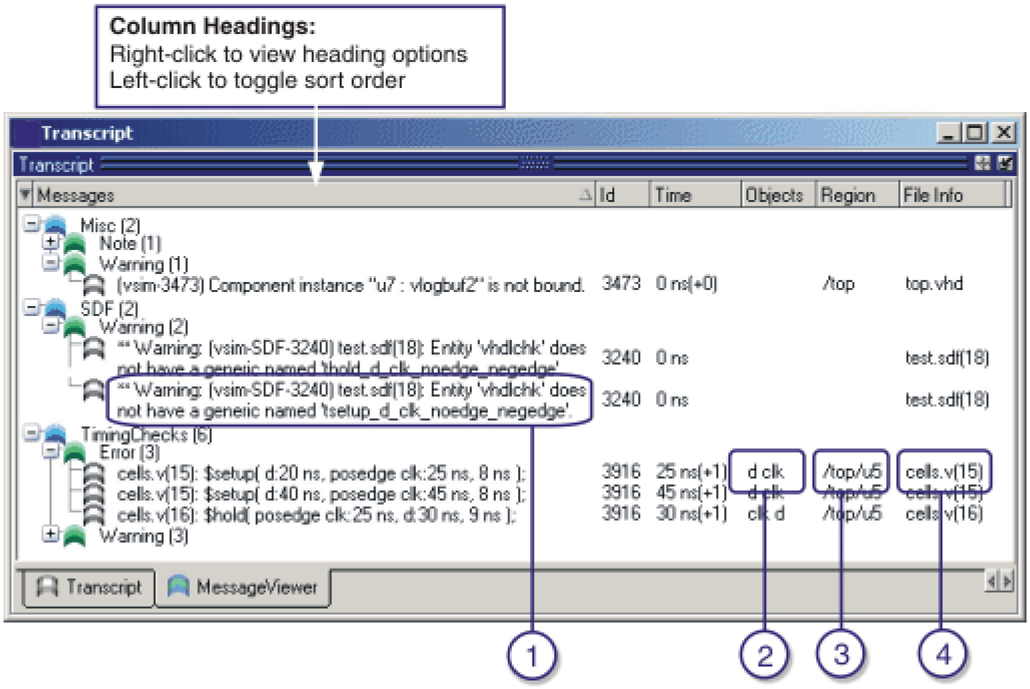
  - both — outputs messages to both the transcript and the WLF file. Default behavior.
  - tran — outputs messages only to the transcript, therefore they are not available in the Message Viewer.
  - wlf — outputs messages only to the WLF file/Message Viewer, therefore they are not available in the transcript.

- **modelsim.ini File** — The msgmode variable in the modelsim.ini file accepts the same values described above for the -msgmode argument.

Message Viewer Interface and Tasks

The Message Viewer tab does not display by default. You can bring it up after a simulation run with the **View** > **Windows** > **Message Viewer** menu item. Figure 2-1 and Table 2-3 provide an overview of the Message Viewer and several tasks you can perform.

Figure 2-1. Message Viewer Tab



Column Headings:
Right-click to view heading options
Left-click to toggle sort order

Messages	Id	Time	Objects	Region	File Info
Misc (2)					
Note (1)					
Warning (1)					
(vsim-3473) Component instance "u7 : vlogbuf2" is not bound.	3473	0 ns(+0)		/top	top.vhd
SDF (2)					
Warning (2)					
** Warning: (vsim-SDF-3240) test.sdf(18): Entity 'vhdlchk' does not have a generic named 'thold_d_clk_noedge_negedge'.	3240	0 ns			test.sdf(18)
** Warning: (vsim-SDF-3240) test.sdf(18): Entity 'vhdlchk' does not have a generic named 'tsetup_d_clk_noedge_negedge'.	3240	0 ns			test.sdf(18)
TimingChecks (6)					
Error (3)					
cells.v(15): $setup(d:20 ns, posedge clk:25 ns, 8 ns);	3916	25 ns(+1)	d clk	/top/u5	cells.v(15)
cells.v(15): $setup(d:40 ns, posedge clk:45 ns, 8 ns);	3916	45 ns(+1)	d clk	/top/u5	cells.v(15)
cells.v(16): $hold(posedge clk:25 ns, d:30 ns, 9 ns);	3916	30 ns(+1)	clk d	/top/u5	cells.v(16)
Warning (3)					

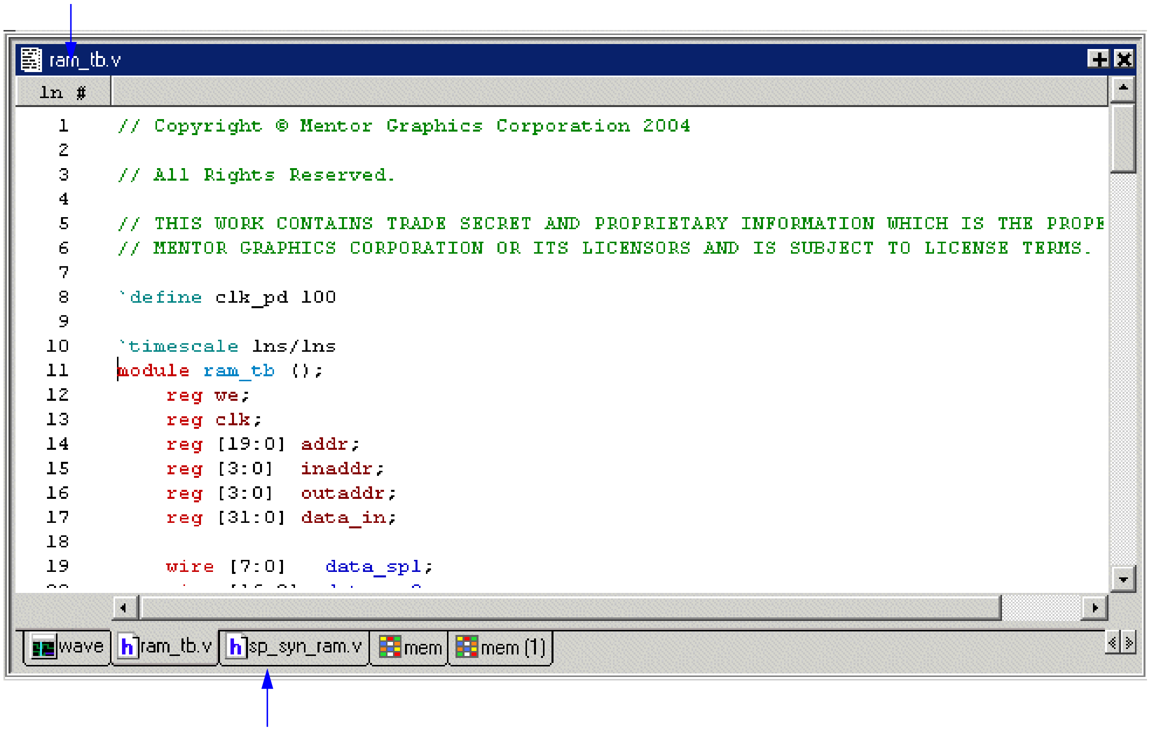
Table 2-3. Message Viewer Tasks

Icon	Task	Action
1	Display a detailed description of the message.	right click the message text then select **View Verbose Message**.
2	Open the source file and add a bookmark to the location of the object(s).	double click the object name(s).
3	Change the focus of the Workspace and Objects panes.	double click the hierarchical reference.
4	Open the source file and set a marker at the line number.	double click the file name.

Multiple Document Interface (MDI) Frame

The MDI frame is an area in the Main window where source editor, memory content, wave, and list windows display. The frame allows multiple windows to be displayed simultaneously, as shown below. A tab appears for each window.

Object name



```
// Copyright © Mentor Graphics Corporation 2004

// All Rights Reserved.

// THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE PROPE
// MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.

`define clk_pd 100

`timescale 1ns/1ns
module ram_tb ();
    reg we;
    reg clk;
    reg [19:0] addr;
    reg [3:0]  inaddr;
    reg [3:0]  outaddr;
    reg [31:0] data_in;

    wire [7:0]   data_spl;
```

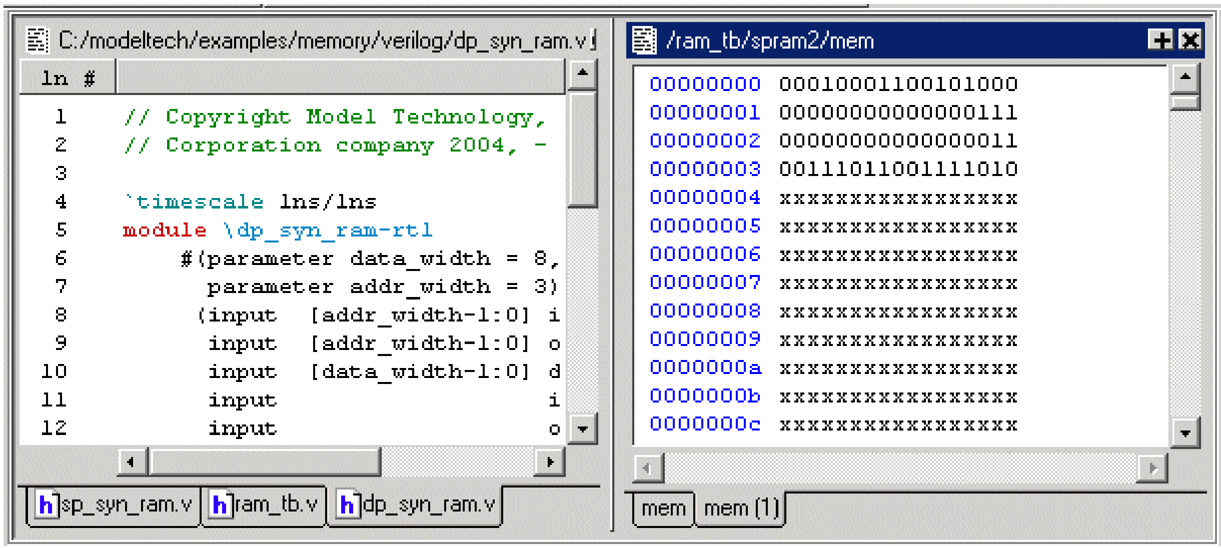
Window tabs

The object name is displayed in the title bar at the top of the window. You can switch between the windows by clicking on a tab.

Organizing Windows with Tab Groups

The MDI can quickly become unwieldy if many windows are open. You can create "tab groups" to help organize the windows. A tab group is a collection of tabs that are separated from other groups of tabs.

The graphic below shows how the collection of files in the picture above could be organized into two tab groups.



```
// Copyright Model Technology,
// Corporation company 2004, -

`timescale 1ns/1ns
module \dp_syn_ram-rtl
        #(parameter data_width = 8,
          parameter addr_width = 3)
         (input   [addr_width-1:0] i
          input   [addr_width-1:0] o
          input   [data_width-1:0] d
          input                    i
          input                    o
```

```
00000000 00010001100101000
00000001 00000000000000111
00000002 00000000000000011
00000003 00111011001111010
00000004 xxxxxxxxxxxxxxxxx
00000005 xxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxx
00000007 xxxxxxxxxxxxxxxxx
00000008 xxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxx
0000000a xxxxxxxxxxxxxxxxx
0000000b xxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxx
```

The commands for creating and organizing tab groups are accessed by right-clicking on any window tab. The table below describes the commands associated with tab groups:

Table 2-4.

Command	Description
New Tab Group	Creates a new tab group containing the selected tab
Move Next Group	Moves the selected tab to the next group in the MDI
Move Prev Group	Moves the selected tab to the previous group in the MDI
View > Vertical / Horizontal	Arranges tab groups top-to-bottom (vertical) or right-to-left (horizontal)

Note that you can also move the tabs within a tab group by dragging them with the middle mouse button.

Navigating in the Main Window

The Main window can contain of a number of "panes" and sub-windows that display various types of information about your design, simulation, or debugging session. Here are a few important points to keep in mind about the Main window interface:

- Windows/panes can be resized, moved, zoomed, undocked, etc. and the changes are persistent.

You have a number of options for re-sizing, re-positioning, undocking/redocking, and generally modifying the physical characteristics of windows and panes.

Windows and panes can be undocked from the main window by pressing the Undock button in the header or by using the **view -undock <window_name>** command. For example, **view -undock objects** will undock the Objects window. The default docked or undocked status of each window or pane can be set with the **PrefMain(ViewUnDocked) <window_name>** preference variable.

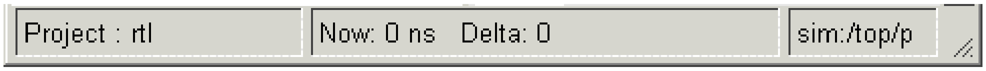
When you exit ModelSim, the current layout is saved so that it appears the same the next time you invoke the tool.

- Menus are context sensitive.

  The menu items that are available and how certain menu items behave depend on which pane or window is active. For example, if the sim tab in the Workspace is active and you choose Edit from the menu bar, the Clear command is disabled. However, if you click in the Transcript pane and choose Edit, the Clear command is enabled. The active pane is denoted by a blue title bar.

For more information, see Navigating the Graphic User Interface.

Main Window Status Bar



Project : rtl

Now: 0 ns   Delta: 0

sim:/top/p

Fields at the bottom of the Main window provide the following information about the current simulation:

Table 2-5.

Field	Description
Project	name of the current project
Now	the current simulation time
Delta	the current simulation iteration number
Profile Samples	the number of profile samples collected during the current simulation
Memory	the total memory used during the current simulation
environment	name of the current context (object selected in the active Structure tab of the Workspace)
line/column	line and column numbers of the cursor in the active Source window

Main Window Toolbar

Buttons on the Main window toolbar give you quick access to various ModelSim commands and functions.

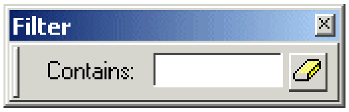
Table 2-6.

Button	Menu equivalent	Command equivalents
New File create a new source file	File > New > Source	
Open open the Open File dialog	File > Open	
Save save the contents of the active pane	File > Save	
Print open the Print dialog	File > Print	
Cut cut the selected text to the clipboard	Edit > Cut	
Copy copy the selected text to the clipboard	Edit > Copy	
Paste paste the clipboard text	Edit > Paste	
Undo undo the last edit	Edit > Undo	
Redo redo the last undone edit	Edit > Redo	
Find find text in the active window	Edit > Find	
Collapse All collapse all instances in the active window	Edit > Expand > Collapse All	
Expand All expand all instance in the active window	Edit > Expand > Expand All	

Table 2-6.

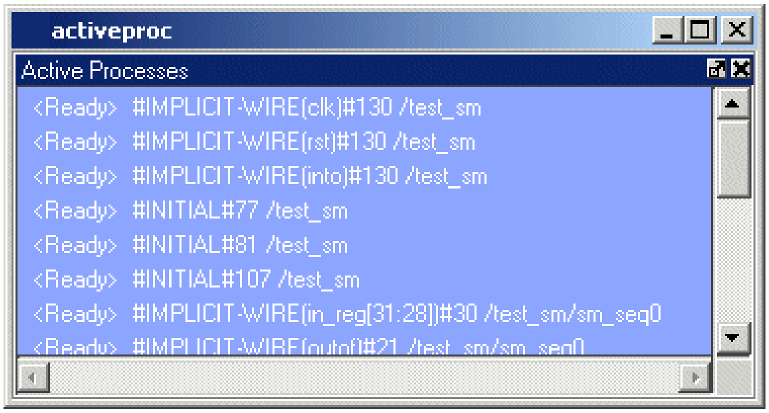
Button	Menu equivalent	Command equivalents
Compile open the Compile Source Files dialog to select files for compilation	Compile > Compile	vcom vlog
Compile All compile all files in the open project	Compile > Compile All	vcom vlog
Simulate load the selected design unit or simulation configuration object	Simulate > Start Simulation	vsim
Break stop the current simulation run	Simulate > Break	
Environment up move up one level in the design hierarchy		
Environment back navigate backward to a previously selected context		
Environment forward navigate forward to a previously selected context		
Restart reload the design elements and reset the simulation time to zero, with the option of maintaining various settings and objects	Simulate > Run > Restart	restart
0 **Run Length** specify the run length for the current simulation	Simulate > Runtime Options	run
Run run the current simulation for the specified run length	Simulate > Run > Run *default_run_length*	run
Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Simulate > Run > Continue	run -continue

Table 2-6.

Button	Menu equivalent	Command equivalents
Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Simulate > Run > Run -All	run -all
Step step the current simulation to the next statement	Simulate > Run > Step	step
Step Over HDL statements are executed but treated as simple statements instead of entered and traced line by line	Simulate > Run > Step -Over	step -over
C Interrupt reactivates the C debugger when stopped in HDL code	Tools > C Debug > C Interrupt	cdbg interrupt
Memory Profiling enable collection of memory usage data	Tools > Profile > Memory	
Performance Profiling enable collection of statistical performance data	Tools > Profile > Performance	
 **Contains** filter items in Objects pane		
Previous Zero Hits jump to previous line with zero coverage		
Next Zero Hits jump to next line with zero coverage		
Show Language Templates display language templates	View > Source > Show Language Templates	

Active Processes Pane

The Active Processes pane displays a list of HDL and SystemC processes. Processes are also displayed in the structure tabs of the Main window Workspace. To filter displayed processes in the structure tabs, select **View > Filter > Processes**.



```
<Ready>  #IMPLICIT-WIRE(clk)#130 /test_sm
<Ready>  #IMPLICIT-WIRE(rst)#130 /test_sm
<Ready>  #IMPLICIT-WIRE(into)#130 /test_sm
<Ready>  #INITIAL#77 /test_sm
<Ready>  #INITIAL#81 /test_sm
<Ready>  #INITIAL#107 /test_sm
<Ready>  #IMPLICIT-WIRE(in_reg[31:28])#30 /test_sm/sm_seq0
<Ready>  #IMPLICIT-WIRE(outof)#21 /test_sm/sm_seq0
```

Process Status

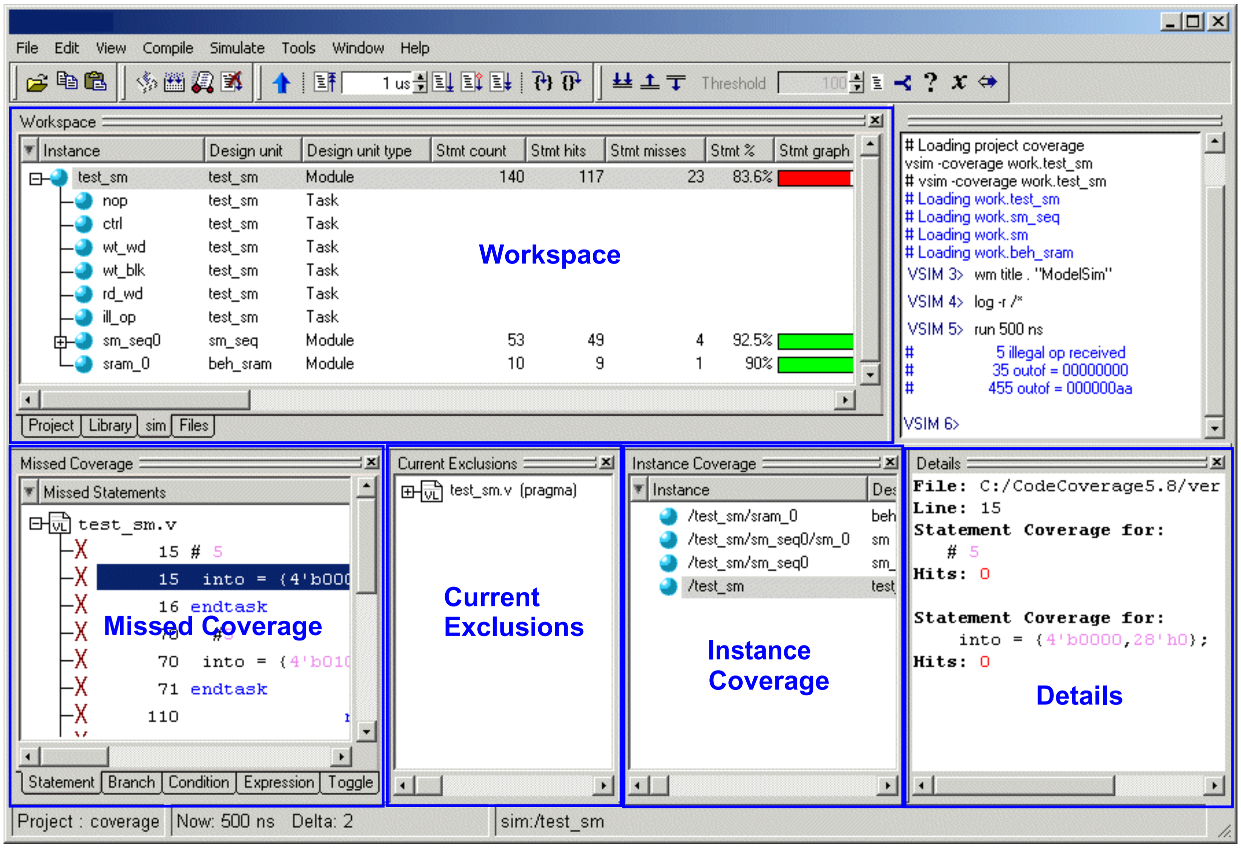
Each object in the scrollbox is preceded by one of the following indicators:

- **<Ready>** — Indicates that the process is scheduled to be executed within the current delta time. If you select a "Ready" process, it will be executed next by the simulator.
- **<Wait>** — Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period. SystemC objects cannot be in a Wait state.
- **<Done>** — Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run. SystemC objects cannot be in a Done state.

Code Coverage Panes

When you run simulations with code coverage enabled, a number of panes display in the Main window. These panes dissect and organize the data collected during coverage analysis. Each pane contains context menus (right-click in the pane to access the menus) with commands appropriate to that pane. You can hide and show the panes by selecting **View > Code Coverage**.

For details about using code coverage refer to Measuring Coverage.



Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt graph
test_sm	test_sm	Module	140	117	23	83.6%	🟥
nop	test_sm	Task					
ctrl	test_sm	Task					
wt_wd	test_sm	Task					
wt_blk	test_sm	Task					
rd_wd	test_sm	Task					
ill_op	test_sm	Task					
sm_seq0	sm_seq	Module	53	49	4	92.5%	🟩
sram_0	beh_sram	Module	10	9	1	90%	🟩

```
# Loading project coverage
vsim -coverage work.test_sm
# vsim -coverage work.test_sm
# Loading work.test_sm
# Loading work.sm_seq
# Loading work.sm
# Loading work.beh_sram
VSIM 3> wm title . "ModelSim"
VSIM 4> log -r /*
VSIM 5> run 500 ns
#              5 illegal op received
#             35 outof = 00000000
#            455 outof = 000000aa
VSIM 6>
```

```
test_sm.v
 X      15 # 5
 X      15  into = {4'b00
 X      16 endtask
 X      70 #
 X      70  into = {4'b01
 X      71 endtask
 X     110
```

```
test_sm.v (pragma)
```

Instance	Des
/test_sm/sram_0	beh
/test_sm/sm_seq0/sm_0	sm
/test_sm/sm_seq0	sm_
/test_sm	test

```
File: C:/CodeCoverage5.8/ver
Line: 15
Statement Coverage for:
   # 5
Hits: 0

Statement Coverage for:
    into = {4'b0000,28'h0};
Hits: 0
```

```
Project : coverage   Now: 500 ns   Delta: 2        sim:/test_sm
```

Workspace Pane

The Workspace pane displays code coverage information in the Files tab and in the structure tabs (e.g., the *sim* tab) that display structure for any datasets being simulated. When coverage is invoked, several columns for displaying coverage data are added to the Workspace pane. You can toggle columns on/off by right-clicking on a column name and selecting from the context menu that appears. The following code coverage-related columns appear in the Workspace pane:

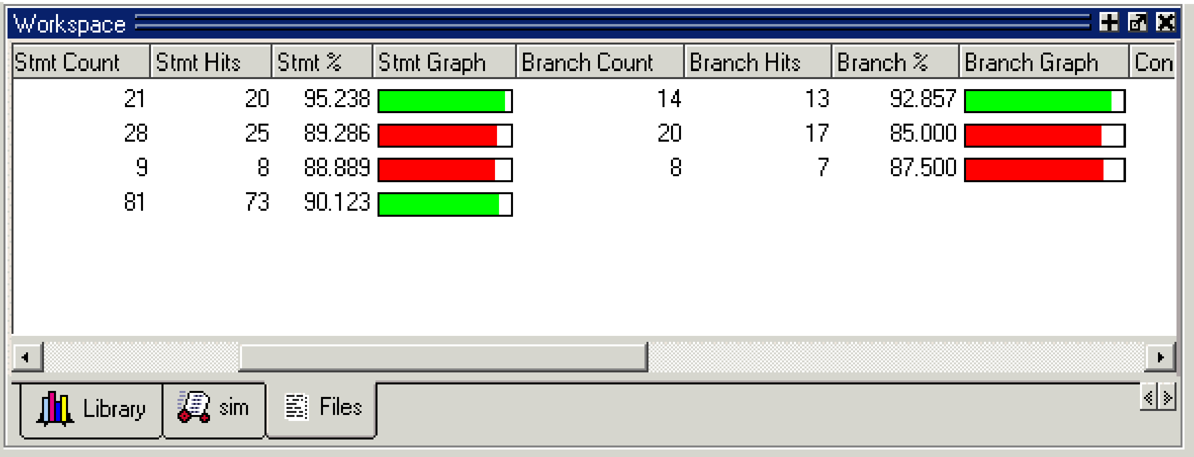
Table 2-7.

Column name	**Description**
Stmt count	in the Files tab, the number of executable statements in each file; in the sim tab, the number of executable statements in each level and all levels under that level
Stmt hits	in the Files tab, the number of executable statements that were executed in each file; in the sim tab, the number of executable statements that were executed in each level and all levels under that level
Stmt misses	in the Files tab, the number of executable statements that were not executed in each file; in the sim tab, the number of executable statements that were not executed in each level and all levels under that level
Stmt %	the current ratio of Stmt hits to Stmt count
Stmt graph	a bar chart displaying the Stmt %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Branch count	in the Files tab, the number of executable branches in each file; in the sim tab, the number of executable branches in each level and all levels under that level
Branch hits	the number of executable branches that have been executed in the current simulation
Branch misses	the number of executable branches that were not executed in the current simulation
Branch %	the current ratio of **Branch** hits to **Branch** count
Branch graph	a bar chart displaying the Branch %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Condition rows	in the Files tab, the number of conditions in each file; in the sim tab, the number of conditions in each level and all levels under that level
Condition hits	in the Files tab, the number of times the conditions in a file have been executed; in the sim tab, the number of times the conditions in a level, and all levels under that level, have been executed
Condition misses	in the Files tab, the number of conditions in a file that were not executed; in the sim tab, the number of conditions in a level, and all levels under that level, that were not executed
Condition %	the current ratio of **Condition** hits to **Condition rows**

Table 2-7.

Column name	**Description**
Condition graph	a bar chart displaying the Condition %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Expression rows	in the Files tab, the number of executable expressions in each file; in the sim tab, the number of executable expressions in each level and all levels subsumed under that level
Expression hits	in the Files tab, the number of times expressions in a file have been executed; in the sim tab, the number of times expressions in a level, and each level under that level, have been executed
Expression misses	in the Files tab, the number of executable expressions in a file that were not executed; in the sim tab, the number of executable expressions in a level, and all levels under that level, that were not executed
Expression %	the current ratio of **Expression** hits to **Expression rows**
Expression graph	a bar chart displaying the Expression %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable
Toggle nodes	the number of points in each instance where the logic will transition from one state to another
Toggle hits	the number of nodes in each instance that have transitioned at least once
Toggle misses	the number of nodes in each instance that have not transitioned at least once
Toggle %	the current ratio of Toggle hits to Toggle nodes
Toggle graph	a bar chart displaying the Toggle %; if the percentage is below 90%, the bar is red; 90% or more, the bar is green; you can change this threshold percentage by editing the **PrefCoverage(cutoff)** preference variable

The diagram below shows a portion of the Workspace window pane with code coverage data displayed.



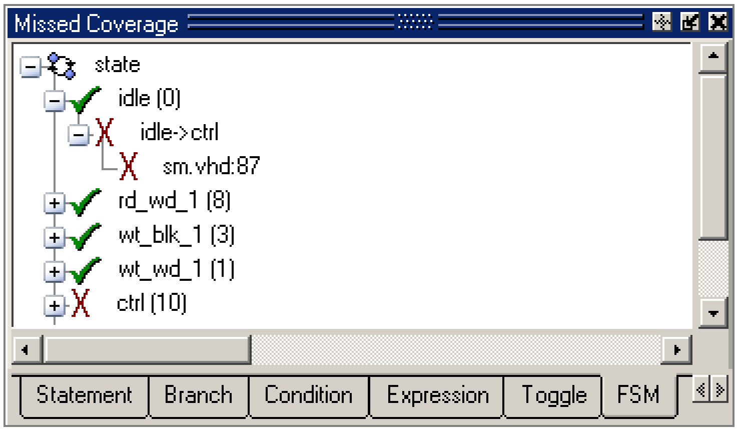
Stmt Count	Stmt Hits	Stmt %	Stmt Graph	Branch Count	Branch Hits	Branch %	Branch Graph	Con
21	20	95.238	🟩	14	13	92.857	🟩	
28	25	89.286	🟥	20	17	85.000	🟥	
9	8	88.889	🟥	8	7	87.500	🟥	
81	73	90.123	🟩					

You can sort code coverage information for any column by clicking the column heading. Clicking the column heading again will reverse the order.

Coverage information in the Workspace pane is dynamically linked to the Missed Coverage pane and the Current Exclusions pane. Click the left mouse button on any file in the Workspace pane to display that file's un-executed statements, branches, conditions, expressions, and toggles in the Missed Coverage pane. Lines from the selected file that are excluded from coverage statistics are displayed in the Current Exclusions pane.

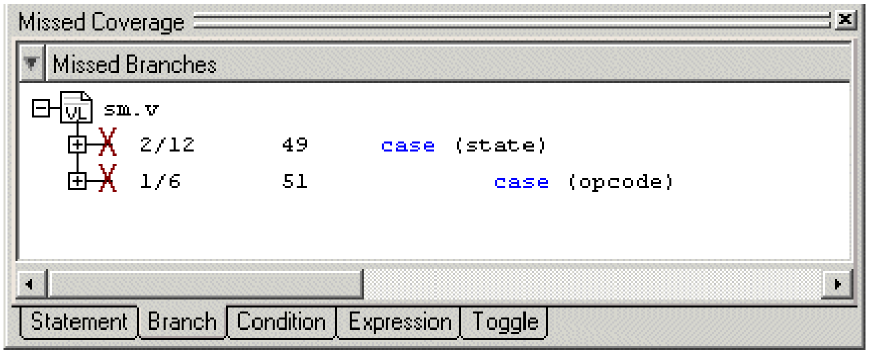
Missed Coverage Pane

When you select a file in the Workspace pane, the Missed Coverage pane displays that uncovered (missed) statements, branches, conditions, and expressions, as well as signals that haven't toggled, and finite state machines (FSM) with uncovered states and transitions. The pane includes a tab for each object, as shown below.



Each tab includes a column for the line number and a column for statement, branch, condition, expression, or toggle on that line. The "X" indicates the object was not executed.

The Branch tab also includes a column for branch code (conditional "if/then/else" and "case" statements). "X_T" indicates that only the true condition of the branch was not executed. "X_F" indicates that only the false condition of the branch was not executed. Fractional numbers indicate how many case statement labels were not executed. For example, if only one of six case labels executed, the Branch tab would indicate "X 1/6."

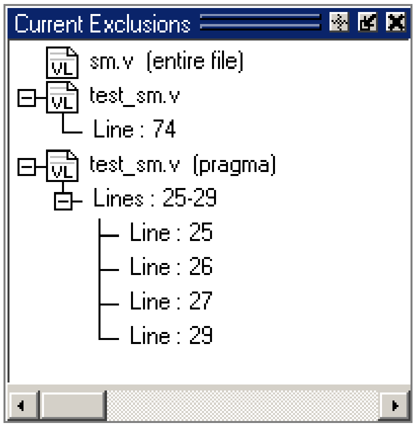


```
sm.v
  X 2/12    49      case (state)
  X 1/6     51              case (opcode)
```

When you right-click any object in the Statement, Branch, Condition, or Expression tabs you can select **Exclude Selection** or **Exclude Selection for Instance <name>** to exclude the object from coverage statistics and make it appear in the Current Exclusions pane.

Current Exclusions Pane

The Current Exclusions pane lists all files and lines that are excluded from coverage statistics. See Excluding Objects from Coverage for more details.



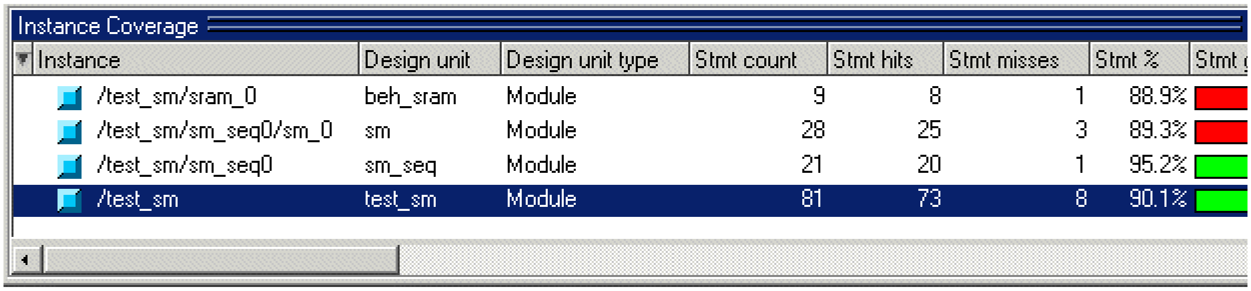
```
sm.v  (entire file)
test_sm.v
  Line : 74
test_sm.v  (pragma)
  Lines : 25-29
    Line : 25
    Line : 26
    Line : 27
    Line : 29
```

The pane does not display by default. Select **View > Code Coverage > Current Exclusions** to display the it.

Instance Coverage Pane

The Instance Coverage pane displays coverage statistics for each instance in a flat, non-hierarchical view. It allows sorting of data columns to be more meaningful, and not confused by hierarchy. The Instance Coverage pane contains the same code coverage statistics columns as in the Workspace pane.

A partial view of the Instance Coverage pane is shown below.

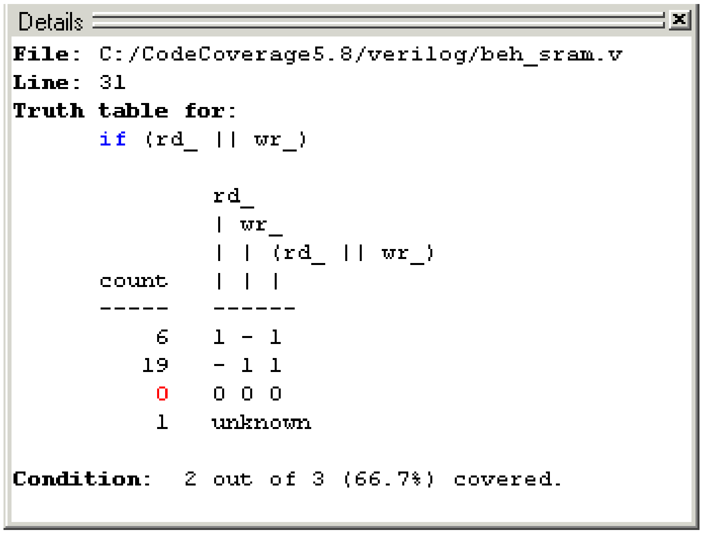


Instance Coverage

Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt g
/test_sm/sram_0	beh_sram	Module	9	8	1	88.9%	🟥
/test_sm/sm_seq0/sm_0	sm	Module	28	25	3	89.3%	🟥
/test_sm/sm_seq0	sm_seq	Module	21	20	1	95.2%	🟩
/test_sm	test_sm	Module	81	73	8	90.1%	🟩

Details Pane

After code coverage is invoked and the simulation is loaded and run, you can turn on the Details pane by selecting **View** > **Code Coverage** > **Details**. The Details pane shows the details of missed coverage. When an object is selected in the Missed Coverage pane, the details of that coverage are displayed in the Details pane. Truth tables will be displayed for condition and expression coverage, as shown here.

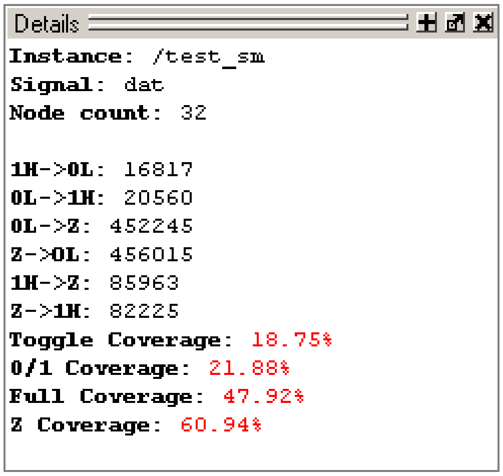


Details

```
File: C:/CodeCoverage5.8/verilog/beh_sram.v
Line: 31
Truth table for:
      if (rd_ || wr_)

              rd_
              | wr_
              | | (rd_ || wr_)
      count   | | |
      -----   ------
          6   1 - 1
         19   - 1 1
          0   0 0 0
          1   unknown

Condition:  2 out of 3 (66.7%) covered.
```

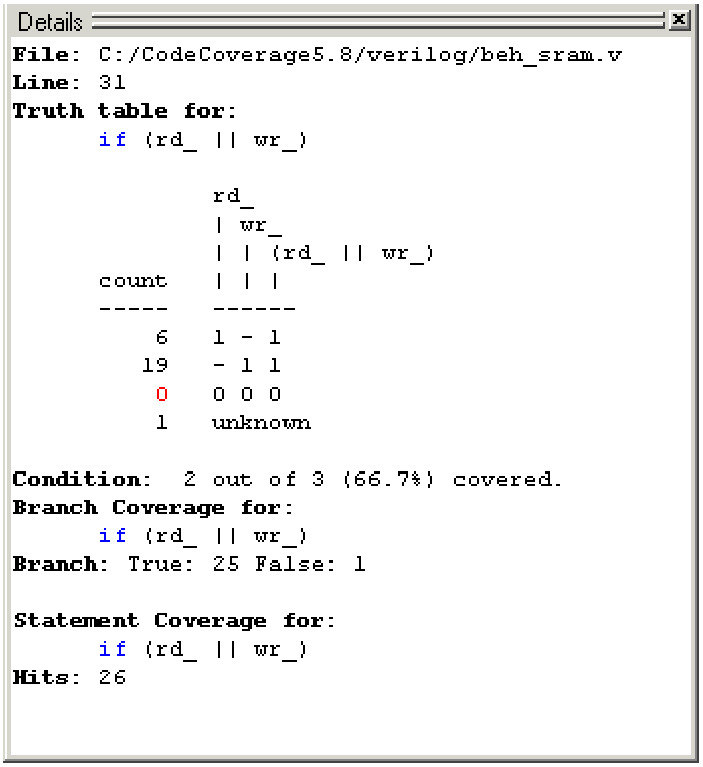
For a description of these truth tables, see Coverage Statistics Details.

Toggle details are displayed as follows:



```
Instance: /test_sm
Signal: dat
Node count: 32

1H->0L: 16817
0L->1H: 20560
0L->Z: 452245
Z->0L: 456015
1H->Z: 85963
Z->1H: 82225
Toggle Coverage: 18.75%
0/1 Coverage: 21.88%
Full Coverage: 47.92%
Z Coverage: 60.94%
```

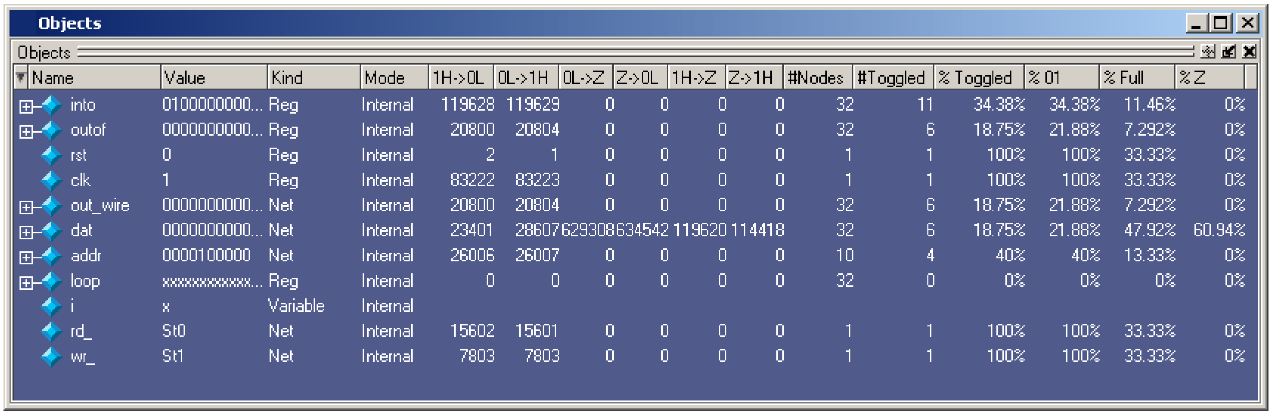
By clicking the left mouse button on the statement Hits column in the Source window, all coverage information for that line will be displayed in the Details pane as shown here:



```
File: C:/CodeCoverage5.8/verilog/beh_sram.v
Line: 31
Truth table for:
      if (rd_ || wr_)

              rd_
              | wr_
              | | (rd_ || wr_)
      count   | | |
      -----   ------
          6   1 - 1
         19   - 1 1
          0   0 0 0
          1   unknown

Condition:  2 out of 3 (66.7%) covered.
Branch Coverage for:
      if (rd_ || wr_)
Branch: True: 25 False: 1

Statement Coverage for:
      if (rd_ || wr_)
Hits: 26
```

Objects Pane Toggle Coverage

Toggle coverage data is displayed in the Objects pane in multiple columns, as shown below. There is a column for each of the six transition types.



Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607	62930	8634542	119620	114418	32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	St0	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	St1	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

Right click any column name to toggle a column on or off.

The following table provides a description of the available columns:

Table 2-8.

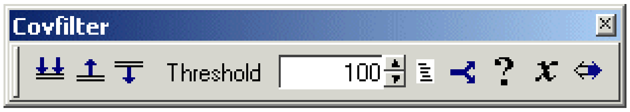
Column name	Description
Name	the name of each object in the current region
Value	the current value of each object
Kind	the object type
Mode	the object mode (internal, in, out, etc.)
1H -> 0L	the number of times each object has transitioned from a 1 or a High state to a 0 or a Low state
0L -> 1H	the number of times each object has transitioned from a 0 or a Low state to 1 or a High state
0L -> Z	the number of times each object has transitioned from a 0 or a Low state to a high impedance (Z) state
Z -> 0L	the number of times each object has transitioned from a high impedance state to a 0 or a Low state
1H -> Z	the number of times each object has transitioned from a 1 or a High state to a high impedance state
Z -> 1H	the number of times each object has transitioned from a high impedance state to 1 or a High state
State Count	the number of values a state machine variable can have
State Hits	the number of state machine variable values that have been hit

Table 2-8.

Column name	Description
State %	the current ration of State Hits to State Count
# Nodes	the number of scalar bits in each object
# Toggled	the number of nodes that have transitioned at least once
% Toggled	the current ratio of the # Toggled to the # Nodes for each object
% 01	the percentage of **1H -> 0L** and **0L -> 1H** transitions that have occurred (transitions in the first two columns)
% Full	the percentage of all transitions that have occurred (all six columns)
% Z	the percentage of **0L -> Z**, **Z -> 0L**, **1H -> Z**, and **Z -> 1H** transitions that have occurred (last four columns)

Code Coverage Toolbar

When you simulate with code coverage enabled, the following toolbar is added to the Main window.



Enable Filtering — enables display filtering of coverage statistics in the Workspace and Instance Coverage panes of the Main window

Threshold above — displays all coverage statistics above the Filter Threshold for selected columns

Threshold below — displays all coverage statistics below the Filter Threshold for selected columns

Filter Threshold — specifies the display coverage percentage for the selected coverage columns

Statement — applies the display filter to all Statement coverage columns in the Workspace and Instance Coverage panes of the Main window

Branch — applies the display filter to all Branch coverage columns in the Workspace and Instance Coverage panes of the Main window

Condition — applies the display filter to all Condition coverage columns in the Workspace and Instance Coverage panes of the Main window

Expression — applies the display filter to all Expression coverage columns in the Workspace and Instance Coverage panes of the Main window

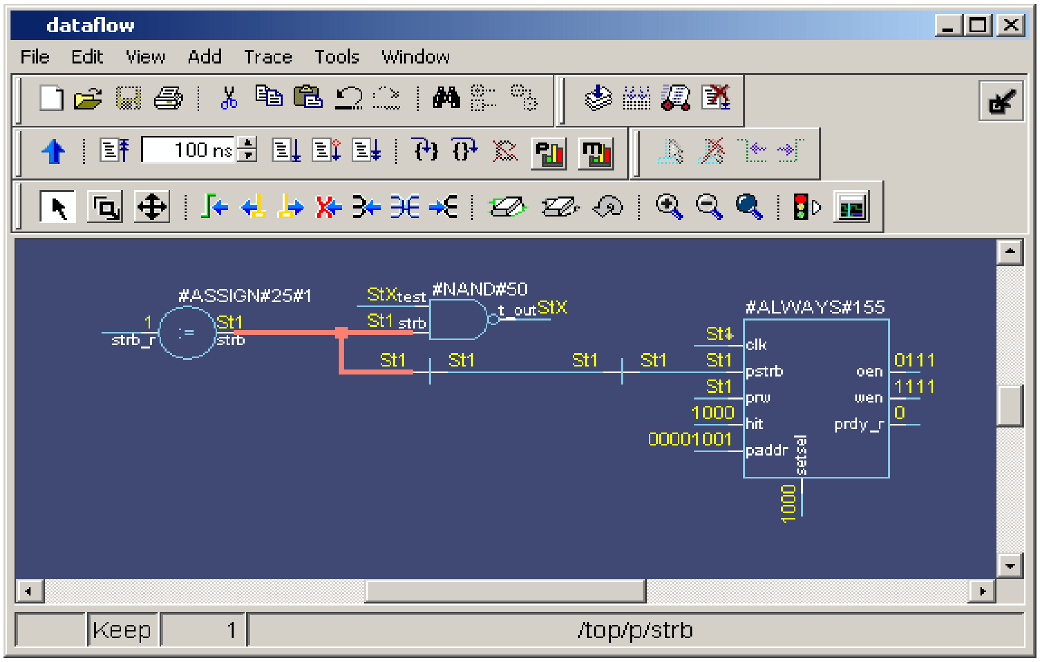
Toggle — applies the display filter to all Toggle coverage columns in the Workspace and Instance Coverage panes of the Main window

Dataflow Window

The Dataflow window allows you to explore the "physical" connectivity of your design. It also allows you to trace events that propagate through the design; and to identify the cause of unexpected outputs.

Note

ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window will show only one process and its attached signals or one signal and its attached processes.



Keep | 1 | /top/p/strb

The Dataflow window displays:

- processes
- signals, nets, and registers

- interconnect

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, PMOS, NMOS, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See Symbol Mapping for details.

Note

You cannot view SystemC objects in the Dataflow window.

Dataflow Window Toolbar

The buttons on the Dataflow window toolbar are described below.

Table 2-9.

Button	Menu equivalent
Print — print the current view of the Dataflow window	File > Print (Windows) File > Print Postscript (UNIX)
Select mode — set left mouse button to select mode and middle mouse button to zoom mode	View > Select
Zoom mode — set left mouse button to zoom mode and middle mouse button to pan mode	View > Zoom
Pan mode — set left mouse button to pan mode and middle mouse button to zoom mode	View > Pan
Cut — cut the selected object(s)	Edit > Cut
Copy — copy the selected object(s)	Edit > Copy
Paste — paste the previously cut or copied object(s)	Edit > Paste
Undo — undo the last action	Edit > Undo
Redo — redo the last undone action	Edit > Redo

Table 2-9.

Button	Menu equivalent
Find — search for an instance or signal	Edit > Find
Trace input net to event — move the next event cursor to the next input event driving the selected output	Trace > Trace next event
Trace Set — jump to the source of the selected input event	Trace > Trace event set
Trace Reset — return the next event cursor to the selected output	Trace > Trace event reset
Trace net to driver of X — step back to the last driver of an unknown value	Trace > TraceX
Expand net to all drivers — display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
Expand net to all drivers and readers — display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net
Expand net to all readers — display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers
Erase highlight — clear the green highlighting which identifies the path you've traversed through the design	Edit > Erase highlight
Erase all — clear the window	Edit > Erase all
Regenerate — clear and redraw the display using an optimal layout	Edit > Regenerate
Zoom In — zoom in by a factor of two from current view	none
Zoom Out — zoom out by a factor of two from current view	none

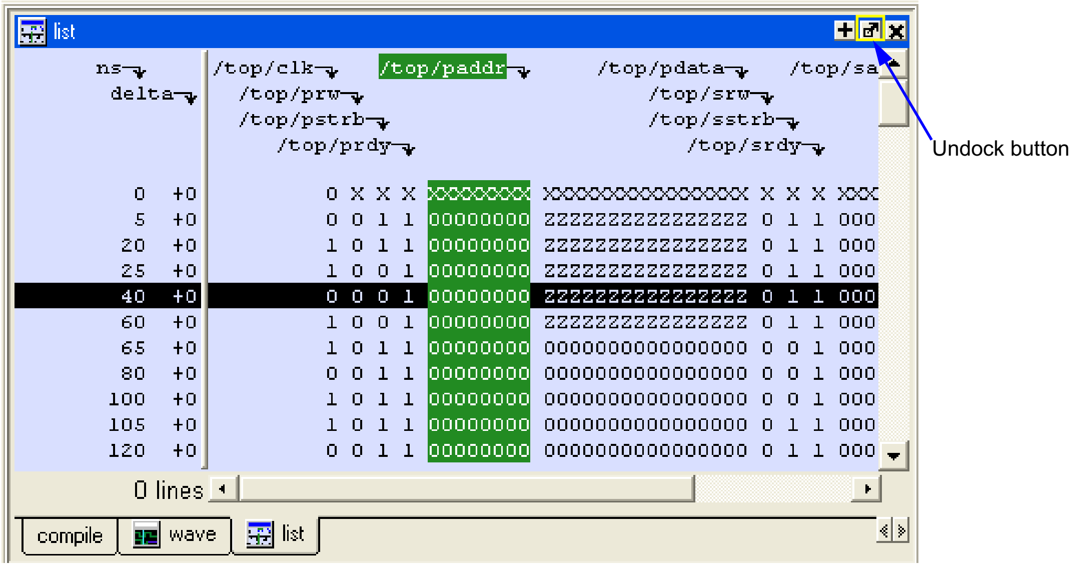
Table 2-9.

Button	Menu equivalent
Zoom Full — zoom out to show all components in the window	none
Stop Drawing — halt any drawing currently happening in the window	none
Show Wave — display the embedded wave viewer pane	View > Show Wave

List Window

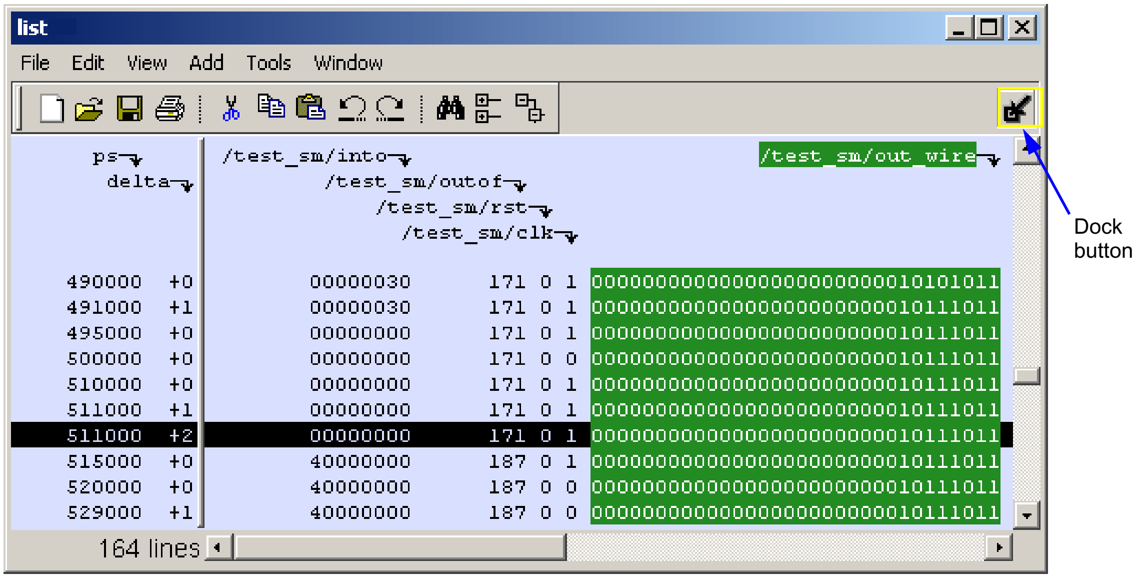
The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable columns, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.

The List window opens by default in the MDI frame of the Main window as shown below.



```
     ns   /top/clk   /top/paddr       /top/pdata      /top/sa
  delta     /top/prw                     /top/srw
            /top/pstrb                   /top/sstrb
               /top/prdy                    /top/srdy

   0  +0         0 X X X  XXXXXXXX  XXXXXXXXXXXXXXXX  X X X  XXX
   5  +0         0 0 1 1  00000000  ZZZZZZZZZZZZZZZZ  0 1 1  000
  20  +0         1 0 1 1  00000000  ZZZZZZZZZZZZZZZZ  0 1 1  000
  25  +0         1 0 0 1  00000000  ZZZZZZZZZZZZZZZZ  0 1 1  000
  40  +0         0 0 0 1  00000000  ZZZZZZZZZZZZZZZZ  0 1 1  000
  60  +0         1 0 0 1  00000000  ZZZZZZZZZZZZZZZZ  0 1 1  000
  65  +0         1 0 1 1  00000000  0000000000000000  0 0 1  000
  80  +0         0 0 1 1  00000000  0000000000000000  0 0 1  000
 100  +0         1 0 1 1  00000000  0000000000000000  0 0 1  000
 105  +0         1 0 1 1  00000000  0000000000000000  0 1 1  000
 120  +0         0 0 1 1  00000000  0000000000000000  0 1 1  000
```

The window can be undocked from the main window by pressing the Undock button in the window header or by using the **view -undock list** command.



```
       ps   /test_sm/into                        /test_sm/out_wire
    delta        /test_sm/outof
                      /test_sm/rst
                         /test_sm/clk

 490000  +0     00000030     171 0 1 00000000000000000000000010101011
 491000  +1     00000030     171 0 1 00000000000000000000000010111011
 495000  +0     00000000     171 0 1 00000000000000000000000010111011
 500000  +0     00000000     171 0 0 00000000000000000000000010111011
 510000  +0     00000000     171 0 1 00000000000000000000000010111011
 511000  +1     00000000     171 0 1 00000000000000000000000010111011
 511000  +2     00000000     171 0 1 00000000000000000000000010111011
 515000  +0     40000000     187 0 1 00000000000000000000000010111011
 520000  +0     40000000     187 0 0 00000000000000000000000010111011
 529000  +1     40000000     187 0 0 00000000000000000000000010111011
```

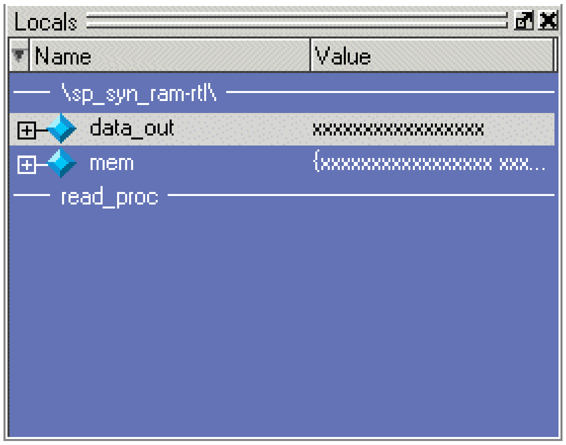
The following type of objects can be viewed in the List pane:

- VHDL — signals, aliases, process variables, and shared variables
- Verilog — nets, registers, and variables
- SystemC — primitive channels and ports
- Comparisons — comparison objects; see Waveform Compare for more information
- Virtuals — Virtual signals and functions

Locals Pane

The Locals pane displays data objects that are immediately visible from the statement that will be executed next (that statement is denoted by a blue arrow in the Source editor window). The contents of the window change from one statement to the next.

The Locals pane includes two columns. The first column lists the names of the immediately visible data objects. The second column lists the current value(s) associated with each name.

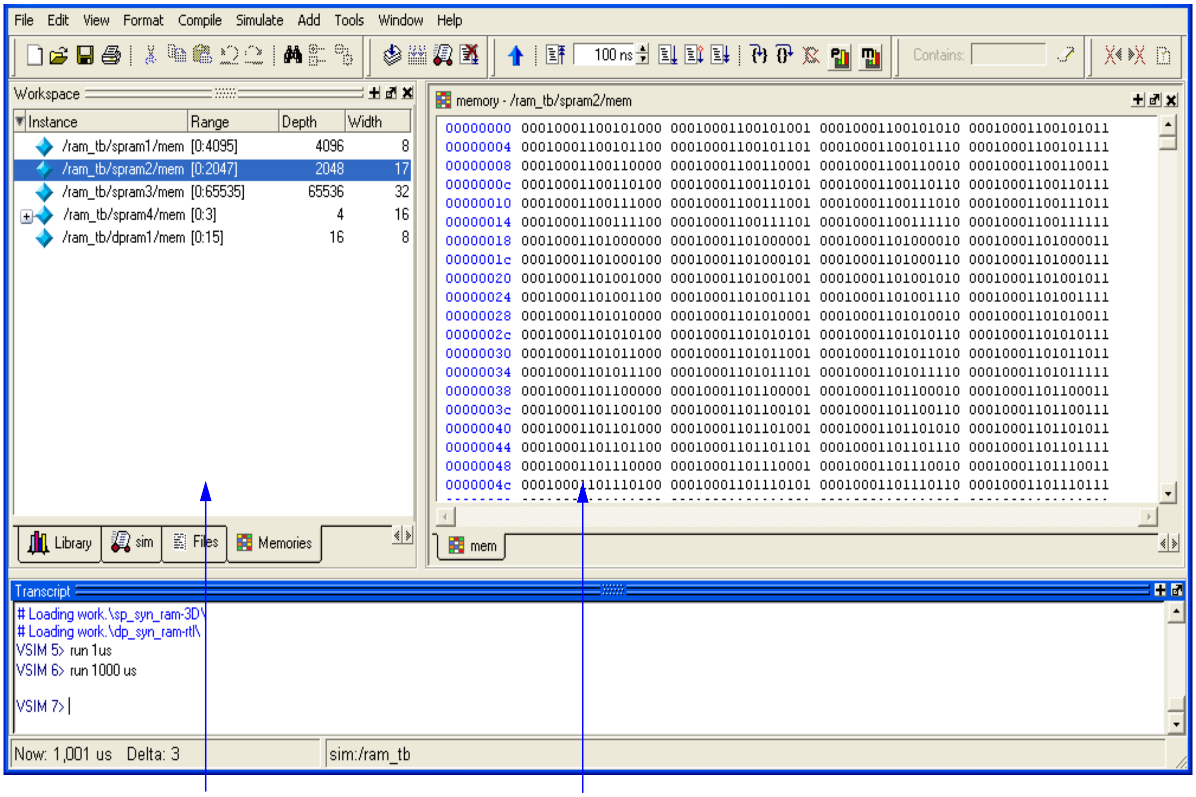


Locals

Name	Value
\sp_syn_ram-rtl\	
⊞ data_out	xxxxxxxxxxxxxxxx
⊞ mem	{xxxxxxxxxxxxxxxx xxx...
read_proc	

Memory Panes

The Main window lists all memories in your design in the Memories tab of the Main window Workspace and displays the contents of a selected memory in the Main window MDI frame.



Instance	Range	Depth	Width
/ram_tb/spram1/mem	[0:4095]	4096	8
/ram_tb/spram2/mem	[0:2047]	2048	17
/ram_tb/spram3/mem	[0:65535]	65536	32
/ram_tb/spram4/mem	[0:3]	4	16
/ram_tb/dpram1/mem	[0:15]	16	8

memory - /ram_tb/spram2/mem

```
00000000 0001000110010100 0 0001000110010100 1 0001000110010101 0 0001000110010101 1
```

```
00000000 00010001100101000 00010001100101001 00010001100101010 00010001100101011
00000004 00010001100101100 00010001100101101 00010001100101110 00010001100101111
00000008 00010001100110000 00010001100110001 00010001100110010 00010001100110011
0000000c 00010001100110100 00010001100110101 00010001100110110 00010001100110111
00000010 00010001100111000 00010001100111001 00010001100111010 00010001100111011
00000014 00010001100111100 00010001100111101 00010001100111110 00010001100111111
00000018 00010001101000000 00010001101000001 00010001101000010 00010001101000011
0000001c 00010001101000100 00010001101000101 00010001101000110 00010001101000111
00000020 00010001101001000 00010001101001001 00010001101001010 00010001101001011
00000024 00010001101001100 00010001101001101 00010001101001110 00010001101001111
00000028 00010001101010000 00010001101010001 00010001101010010 00010001101010011
0000002c 00010001101010100 00010001101010101 00010001101010110 00010001101010111
00000030 00010001101011000 00010001101011001 00010001101011010 00010001101011011
00000034 00010001101011100 00010001101011101 00010001101011110 00010001101011111
00000038 00010001101100000 00010001101100001 00010001101100010 00010001101100011
0000003c 00010001101100100 00010001101100101 00010001101100110 00010001101100111
00000040 00010001101101000 00010001101101001 00010001101101010 00010001101101011
00000044 00010001101101100 00010001101101101 00010001101101110 00010001101101111
00000048 00010001101110000 00010001101110001 00010001101110010 00010001101110011
0000004c 00010001101110100 00010001101110101 00010001101110110 00010001101110111
```

```
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\
VSIM 5> run 1us
VSIM 6> run 1000 us

VSIM 7> |
```

Now: 1,001 us Delta: 3 sim:/ram_tb

Memory list
Memory contents

The memory list is from the top-level of the design. In other words, it is not sensitive to the context selected in the Structure tab.

ModelSim identifies certain kinds of arrays in various scopes as memories. Memory identification depends on the array element kind as well as the overall array kind (i.e. associative array, unpacked array, etc.).

Table 2-10.

	VHDL	Verilog/SystemVerilog
Element kind	enum[1], std_logic_vector, std_bit_vector, or integer.	any integral type. (i.e. integer_type): shortint, int, longint, byte, bit (2 state), logic, reg, integer, time (4 state), packed_struct / packed_union (2 state), packed_struct / packed_union (4 state), packed_array (single-Dim, multi-D, 2 state and 4 state), enum or string.

Table 2-10.

	VHDL	Verilog/SystemVerilog
Scope: recognizable in	architecture, process, or record	module, interface, package, compilation unit, struct, or static variables within a task / function / named block / class
Array kind	single-dimensional or multi-dimensional	any combination of unpacked, dynamic and assotiative arrays[2]

[1] These enumerated type value sets must have values that are longer than one character. The listed width is the number of entries in the enumerated type definition and the depth is the size of the array itself.
[2] Any combination of unpacked, dynamic, and associative arrays is considered a memory, provided the leaf level of the data structure is a string or an integral type.

Associative Arrays in Verilog/SystemVerilog

For an associative array to be recognized as a memory, the index must be of an integral type (see above) or wildcard type.

For associative arrays, the element kind can be any type allowed for fixed-size arrays.

Viewing Single and Multidimensional Memories

Single dimensional arrays of integers are interpreted as 2D memory arrays. In these cases, the word width listed in the Memory List pane is equal to the integer size, and the depth is the size of the array itself.

Memories with three or more dimensions display with a plus sign ’+’ next to their names in the Memory List. Click the ’+’ to show the array indices under that level. When you finally expand down to the 2D level, you can double-click on the index, and the data for the selected 2D slice of the memory will appear in a memory contents pane in the MDI frame.

Viewing Packed Arrays

By default packed dimensions are treated as single vectors in the memory contents pane. To expand packed dimensions of packed arrays, select **View > Memory Contents > Expand Packed Memories**.

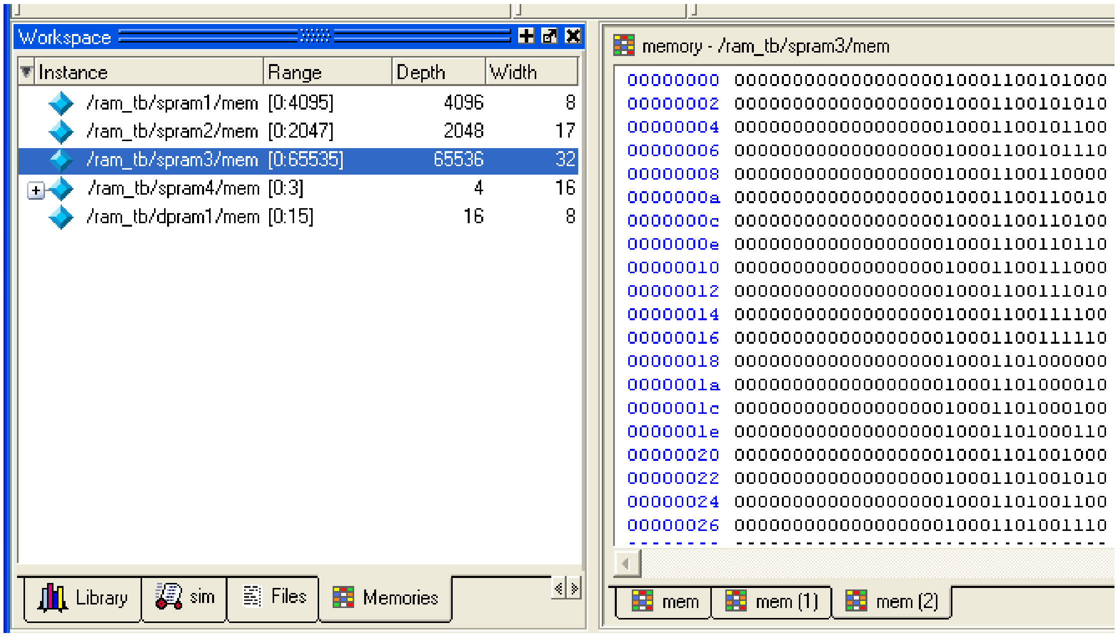
To change the permanent default, edit the PrefMemory(ExpandPackedMem) variable. This variable affects only packed arrays. If the variable is set to 1, the packed arrays are treated as unpacked arrays and are expanded along the packed dimensions such that they appear as a linearized bit vector. See Simulator GUI Preferences for details on setting preference variables.

Viewing Memory Contents

When you double-click an instance on the Memory tab, ModelSim automatically displays a memory contents pane in the MDI frame (see Multiple Document Interface (MDI) Frame). You can also enter the command **add mem <instance>** at the vsim command prompt.

Viewing Multiple Memory Instances

You can view multiple memory instances simultaneously. A memory tab appears in the MDI frame for each instance you double-click in the Memory list.



Instance	Range	Depth	Width
/ram_tb/spram1/mem	[0:4095]	4096	8
/ram_tb/spram2/mem	[0:2047]	2048	17
/ram_tb/spram3/mem	[0:65535]	65536	32
/ram_tb/spram4/mem	[0:3]	4	16
/ram_tb/dpram1/mem	[0:15]	16	8

```
00000000 00000000000000000010001100101000
00000002 00000000000000000010001100101010
00000004 00000000000000000010001100101100
00000006 00000000000000000010001100101110
00000008 00000000000000000010001100110000
0000000a 00000000000000000010001100110010
0000000c 00000000000000000010001100110100
0000000e 00000000000000000010001100110110
00000010 00000000000000000010001100111000
00000012 00000000000000000010001100111010
00000014 00000000000000000010001100111100
00000016 00000000000000000010001100111110
00000018 00000000000000000010001101000000
0000001a 00000000000000000010001101000010
0000001c 00000000000000000010001101000100
0000001e 00000000000000000010001101000110
00000020 00000000000000000010001101001000
00000022 00000000000000000010001101001010
00000024 00000000000000000010001101001100
00000026 00000000000000000010001101001110
```

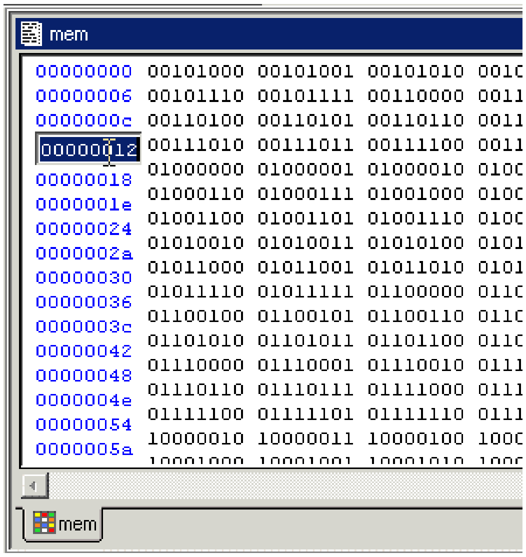
See Organizing Windows with Tab Groups for more information on tabs.

Saving Memory Formats in a DO File

You can save all open memory instances and their formats (e.g., address radix, data radix, etc.) by creating a DO file. With the memory tab active, select **File > Save As**. The Save memory format dialog box opens, where you can specify the name for the saved file. By default it is named *mem.do*. The file will contain all open memory instances and their formats. To load it at a later time, select **File > Load**.

Direct Address Navigation

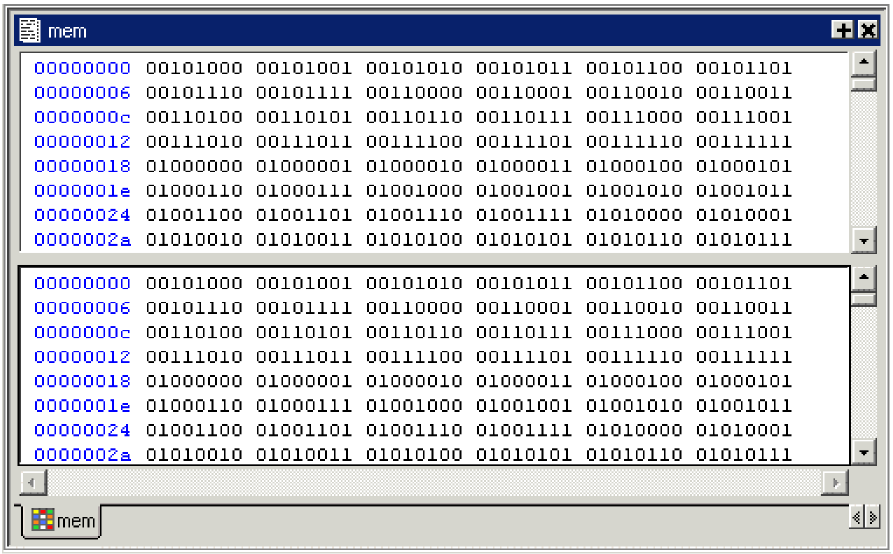
You can navigate to any address location directly by editing the address in the address column. Double-click on any address, type in the desired address, and hit **Enter**. The address display scrolls to the specified location.



```
00000000 00101000 00101001 00101010 0010
00000006 00101110 00101111 00110000 0011
0000000c 00110100 00110101 00110110 0011
00000012 00111010 00111011 00111100 0011
00000018 01000000 01000001 01000010 0100
0000001e 01000110 01000111 01001000 0100
00000024 01001100 01001101 01001110 0100
0000002a 01010010 01010011 01010100 0101
00000030 01011000 01011001 01011010 0101
00000036 01011110 01011111 01100000 0110
0000003c 01100100 01100101 01100110 0110
00000042 01101010 01101011 01101100 0110
00000048 01110000 01110001 01110010 0111
0000004e 01110110 01110111 01111000 0111
00000054 01111100 01111101 01111110 0111
0000005a 10000010 10000011 10000100 1000
         10001000 10001001 10001010 1000
```

Splitting the Memory Contents Pane

To split a memory contents window into two screens displaying the contents of a single memory instance, select **View > Memory Contents > Split Screen** (or right-click in the pane and select **View Contents** from the pop-up menu). This allows you to view different address locations within the same memory instance simultaneously.



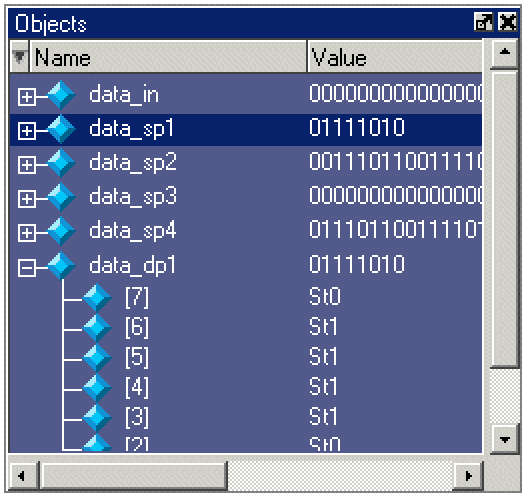
```
00000000 00101000 00101001 00101010 00101011 00101100 00101101
00000006 00101110 00101111 00110000 00110001 00110010 00110011
0000000c 00110100 00110101 00110110 00110111 00111000 00111001
00000012 00111010 00111011 00111100 00111101 00111110 00111111
00000018 01000000 01000001 01000010 01000011 01000100 01000101
0000001e 01000110 01000111 01001000 01001001 01001010 01001011
00000024 01001100 01001101 01001110 01001111 01010000 01010001
0000002a 01010010 01010011 01010100 01010101 01010110 01010111
```

```
00000000 00101000 00101001 00101010 00101011 00101100 00101101
00000006 00101110 00101111 00110000 00110001 00110010 00110011
0000000c 00110100 00110101 00110110 00110111 00111000 00111001
00000012 00111010 00111011 00111100 00111101 00111110 00111111
00000018 01000000 01000001 01000010 01000011 01000100 01000101
0000001e 01000110 01000111 01001000 01001001 01001010 01001011
00000024 01001100 01001101 01001110 01001111 01010000 01010001
0000002a 01010010 01010011 01010100 01010101 01010110 01010111
```

Objects Pane

The Objects pane shows the names and current values of declared data objects in the current region (selected in the structure tabs of the Workspace). Data objects include signals, nets, registers, constants and variables not declared in a process, generics, parameters, and SystemC transactions and member data variables.

Clicking an entry in the window highlights that object in the Dataflow and Wave windows. Double-clicking an entry highlights that object in a Source editor window (opening a Source editor window if one is not open already). You can also right click an object name and add it to the List or Wave window, or the current log file.



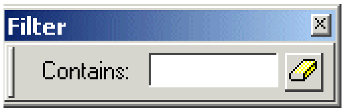
Name	Value
data_in	00000000000000
data_sp1	01111010
data_sp2	00111011001111
data_sp3	00000000000000
data_sp4	01110110011110
data_dp1	01111010
[7]	St0
[6]	St1
[5]	St1
[4]	St1
[3]	St1
[2]	St0

Filtering the Objects List

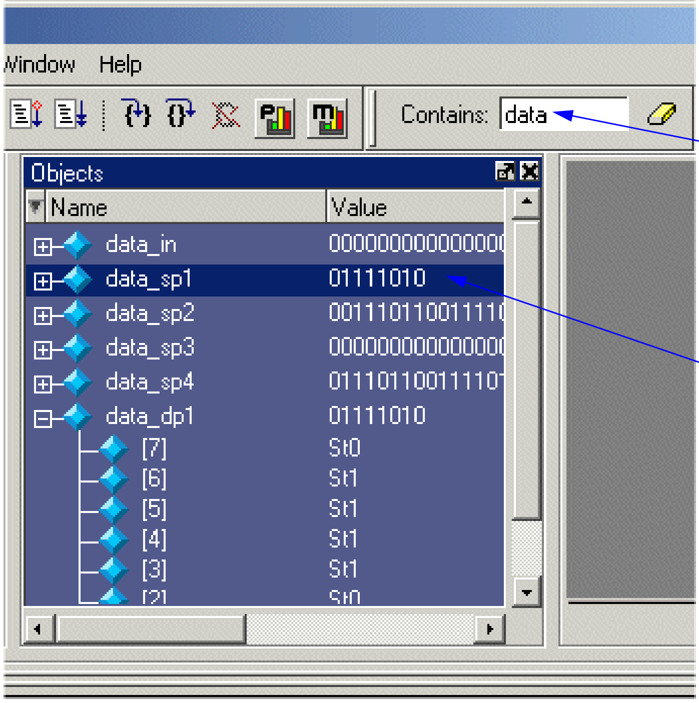
You can filter the objects list by name or by object type.

Filtering by Name

To filter by name, start typing letters in the **Contains** field on the Main window toolbar.



As you type letters, the objects list filters to show only those signals that contain those letters.



Contains: data

Name	Value
data_in	000000000000000
data_sp1	01111010
data_sp2	001110110011110
data_sp3	000000000000000
data_sp4	011101100111101
data_dp1	01111010
[7]	St0
[6]	St1
[5]	St1
[4]	St1
[3]	St1
[2]	St0

As you type letters in the Contains: field...

...the objects list filters dynamically to show only objects that match your entry

To display all objects again, click the Eraser icon to clear the entry.

Filters are stored relative to the region selected in the Structure window. If you re-select a region that had a filter applied, that filter is restored. This allows you to apply different filters to different regions.

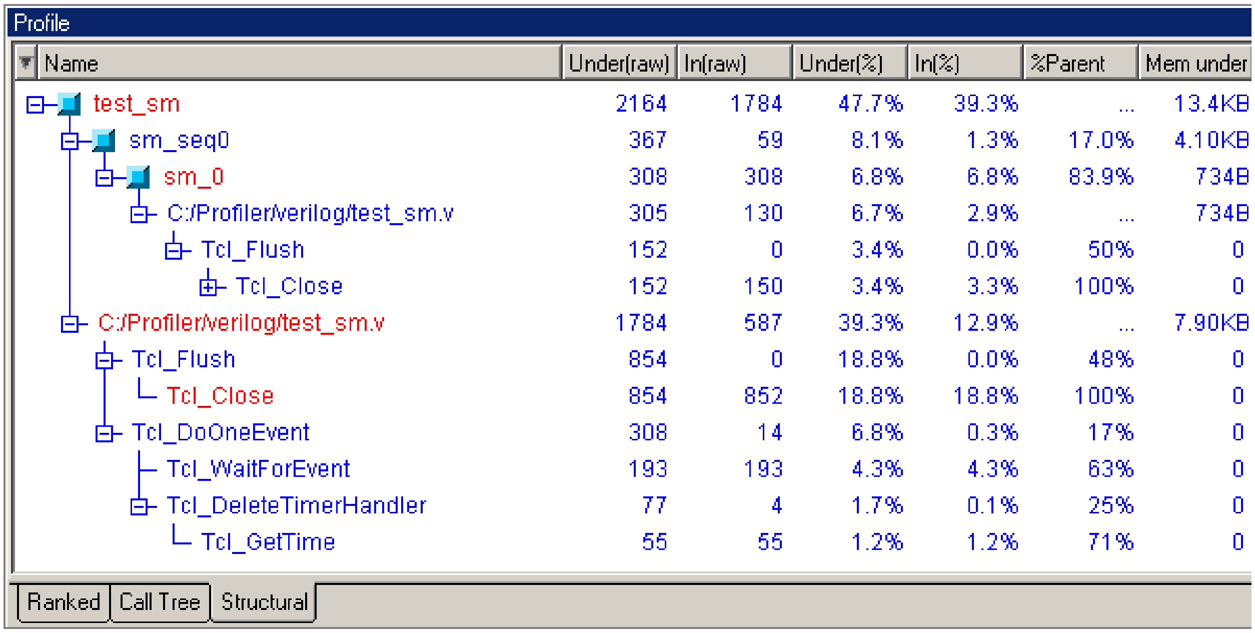
Filtering by Signal Type

The **View** > **Filter** menu selection allows you to specify which signal types to display in the Objects window. Multiple options can be selected.

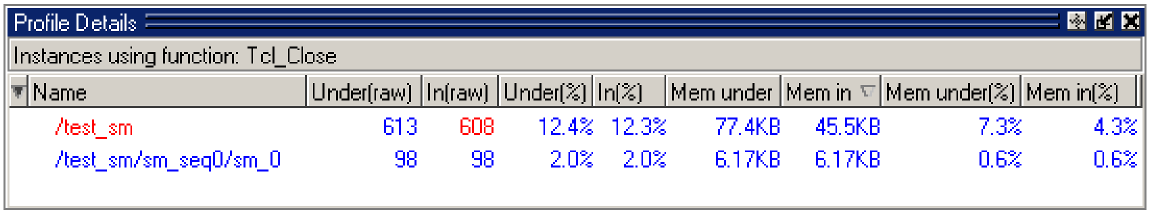
Profile Panes

The Profile and Profile Details panes display the results of statistical performance and memory allocation profiling. By default, both panes are displayed within the Main window but they can be undocked from the Main window to stand alone. Each pane contains three tabs for displaying profile results: Ranked, Call Tree, and Structural.

For details about using the profiler refer to Profiling Performance and Memory Use.



Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent	Mem under
test_sm	2164	1784	47.7%	39.3%	...	13.4KB
sm_seq0	367	59	8.1%	1.3%	17.0%	4.10KB
sm_0	308	308	6.8%	6.8%	83.9%	734B
C:/Profiler/verilog/test_sm.v	305	130	6.7%	2.9%	...	734B
Tcl_Flush	152	0	3.4%	0.0%	50%	0
Tcl_Close	152	150	3.4%	3.3%	100%	0
C:/Profiler/verilog/test_sm.v	1784	587	39.3%	12.9%	...	7.90KB
Tcl_Flush	854	0	18.8%	0.0%	48%	0
Tcl_Close	854	852	18.8%	18.8%	100%	0
Tcl_DoOneEvent	308	14	6.8%	0.3%	17%	0
Tcl_WaitForEvent	193	193	4.3%	4.3%	63%	0
Tcl_DeleteTimerHandler	77	4	1.7%	0.1%	25%	0
Tcl_GetTime	55	55	1.2%	1.2%	71%	0



Instances using function: Tcl_Close

Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
/test_sm	613	608	12.4%	12.3%	77.4KB	45.5KB	7.3%	4.3%
/test_sm/sm_seq0/sm_0	98	98	2.0%	2.0%	6.17KB	6.17KB	0.6%	0.6%

Profile Pane Columns

The Profile panes include the columns described below.

- **Name** — lists the filename of an HDL function or instance, and the line number at which it appears. Most useful names consist of a line of VHDL or Verilog source code. If you use a PLI/VPI or FLI routine, then the name of the C function that implements that routine can also appear in the Name column.
- **Under (raw)** — lists the raw number of Profiler samples collected during the execution of a function, including all support routines under that function; or, the number of

samples collected for an instance, including all instances beneath it in the structural hierarchy.

- **In (raw)** — lists the raw number of Profiler samples collected during a function or instance.
- **Under%** — lists the ratio (as a percentage) of the samples collected during the execution of a function and all support routines under that function to the total number of samples collected; or, the ratio of the samples collected during an instance, including all instances beneath it in the structural hierarchy, to the total number of samples collected.
- **In%** — lists the ratio (as a percentage) of the total samples collected during a function or instance.
- **%Parent** — (not in Ranked view) lists the ratio, as a percentage, of the samples collected during the execution of a function or instance to the samples collected in the parent function or instance.
- **Mem under** — lists the amount of memory allocated to a function, including all support routines under that function; or, the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy.
- **Mem in** — lists the amount of memory allocated to a function or instance.
- **Mem under (%)** — lists the ratio (as a percentage) of the amount of memory allocated to a function and all of its support routines to the total memory available; or, the ratio of the amount of memory allocated to an instance, including all instances beneath it in the structural hierarchy, to the total memory available.
- **Mem in (%)** — lists the ratio (as a percentage) of the amount of memory allocated to a function or instance to the total memory available.
- **%Parent** — lists (not in Ranked view) the ratio, as a percentage, of the memory allocated to a function or instance to the memory allocated to the parent function or instance.

Profiler Toolbar

The Ranked, Call Tree and Structural views all share a toolbar in the Main window. The table below describes the icons in this toolbar.

Table 2-11.

Button	Menu equivalent	Command equivalents
Memory Profiling enable collection of memory usage data	Tools > Profile > Memory	

Table 2-11.

Button	Menu equivalent	Command equivalents
Performance Profiling enable collection of statistical performance data	Tools > Profile > Performance	
Collapse Sections on/off toggling of reporting for collapsed processes and functions.	Tools > Profile > Collapse Sections	
Profile Cutoff display performance and memory profile data equal to or greater than set percentage		
Refresh profile data refresh profile performance and memory data after changing profile cutoff		
Save profile results save profile data to output file (prompts for file name)	Tools > Profile > Profile Report	profile report
Profile Find search for the named string in the Profile pane		

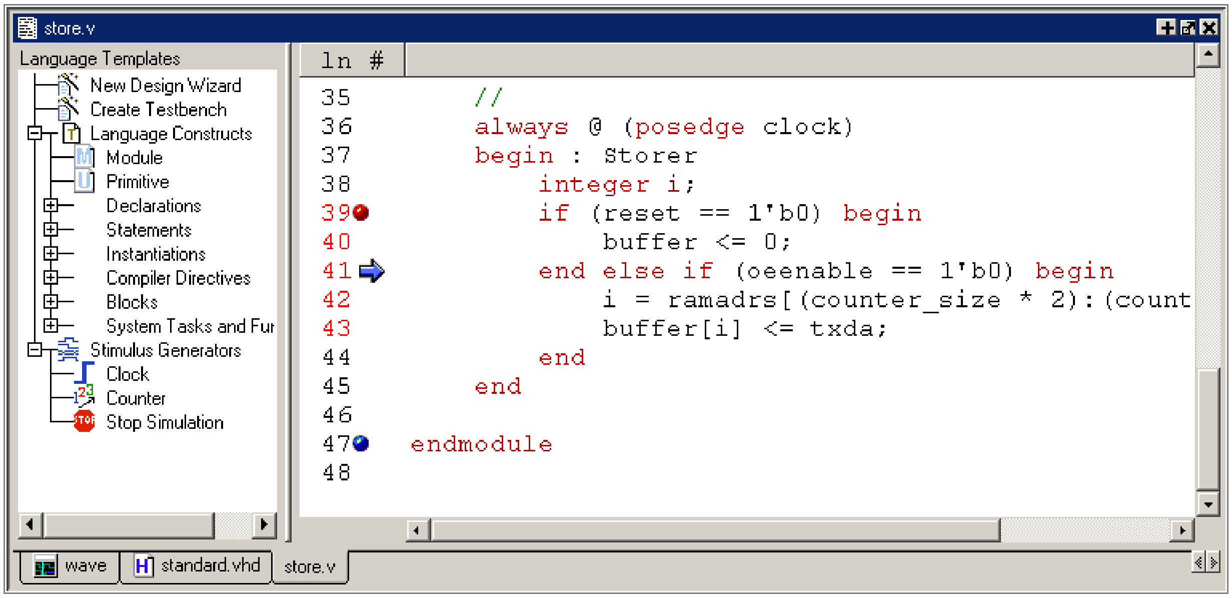
Source Window

Source files display by default in the MDI frame of the Main window. The window can be undocked from the Main window by pressing the Undock button in the window header or by using the **view -undock source** command.

You can edit source files as well as set breakpoints, step through design files, and view code coverage statistics.

By default, the Source window displays your source code with line numbers. You may also see the following graphic elements:

- Red line numbers — denote lines on which you can set a breakpoint
- Blue arrow — denotes the currently active line or a process that you have selected in the Active Processes Pane
- Red circles — denote file-line breakpoints; gray circles denote breakpoints that are currently disabled
- Blue circles — denote line bookmarks
- Language Templates pane — displays Language Templates



```
    //
    always @ (posedge clock)
    begin : Storer
        integer i;
        if (reset == 1'b0) begin
            buffer <= 0;
        end else if (oeenable == 1'b0) begin
            i = ramadrs[(counter_size * 2):(count
            buffer[i] <= txda;
        end
    end

endmodule

```

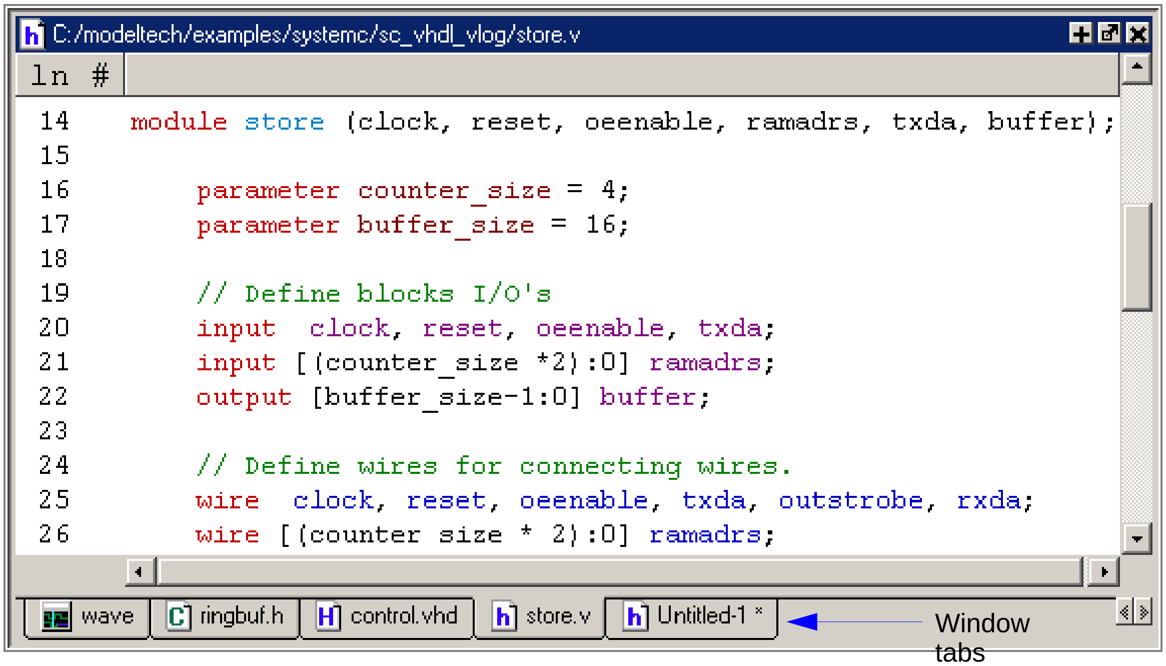
Opening Source Files

You can open source files using the **File > Open** command. Alternatively, you can open source files by double-clicking objects in other windows. For example, if you double-click an item in the Objects window or in the structure tab of the Workspace, the underlying source file for the object will open, and the cursor will scroll to the line where the object is defined.

By default files you open from within the design (e.g., by double-clicking an object in the Objects pane) open in Read Only mode. To make the file editable, right-click in the Source window and select Read Only. To change this default behavior, set the PrefSource(ReadOnly) variable to 0. See Simulator GUI Preferences for details on setting preference variables.

Displaying Multiple Source Files

By default each file you open or create is marked by a window tab, as shown in the graphic below.



```
module store (clock, reset, oeenable, ramadrs, txda, buffer);

    parameter counter_size = 4;
    parameter buffer_size = 16;

    // Define blocks I/O's
    input  clock, reset, oeenable, txda;
    input [(counter_size *2):0] ramadrs;
    output [buffer_size-1:0] buffer;

    // Define wires for connecting wires.
    wire  clock, reset, oeenable, txda, outstrobe, rxda;
    wire [(counter size * 2):0] ramadrs;
```

Window tabs

See Organizing Windows with Tab Groups for more information on these tabs.

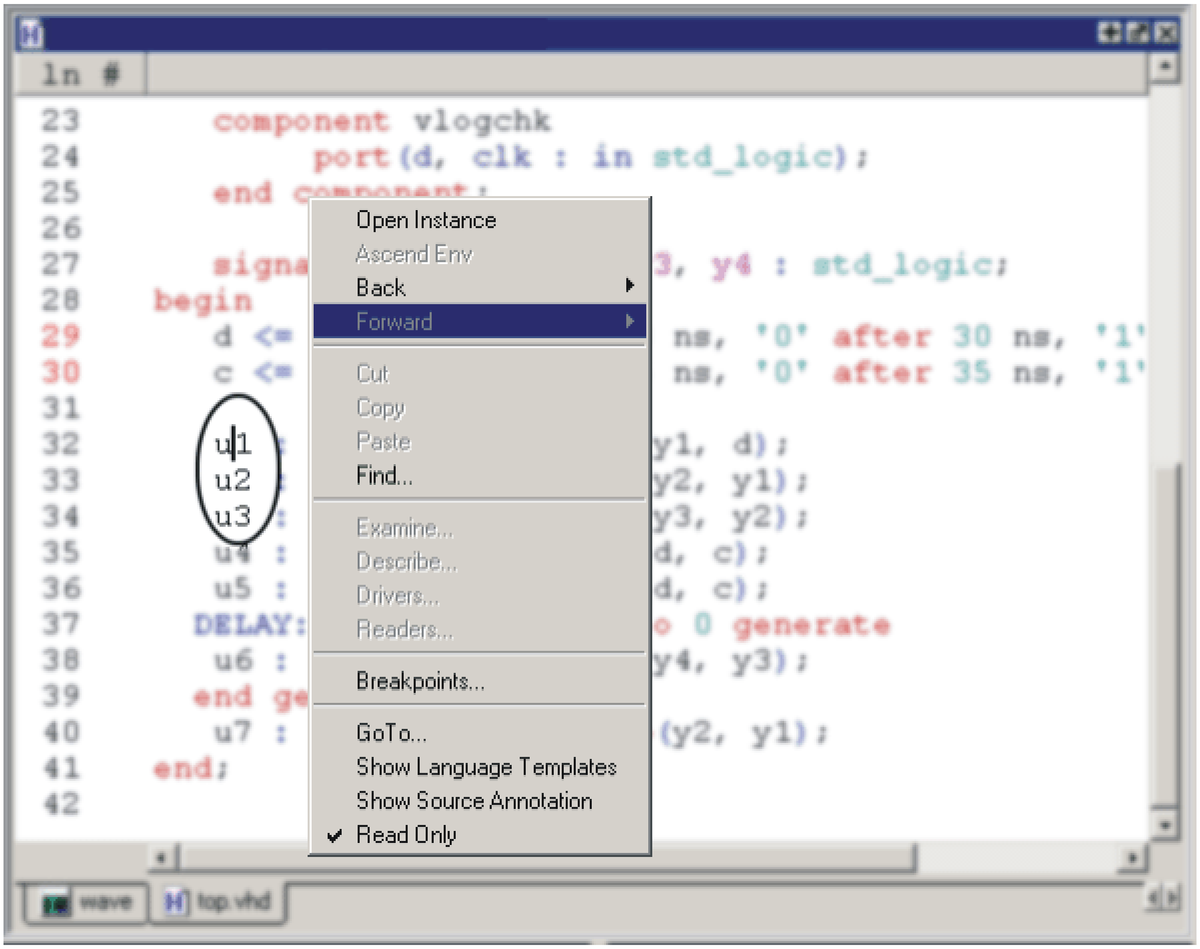
Dragging and Dropping Objects into the Wave and List Windows

ModelSim allows you to drag and drop objects from the Source window to the Wave and List windows. Double-click an object to highlight it, then drag the object to the Wave or List window. To place a group of objects into the Wave and List windows, drag and drop any section of highlighted code.

Setting your Context by Navigating Source Files

When debugging your design from within the GUI, you can change your context while analyzing your source files. Figure 2-2 shows the pop-up menu the tool displays after you select then right-click an instance name in a source file.

Figure 2-2. Setting Context from Source Files



```
        component vlogchk
             port (d, clk : in std_logic);
        end component;

        signa                               3, y4 : std_logic;
     begin
        d <=                                ns, '0' after 30 ns, '1'
        c <=                                ns, '0' after 35 ns, '1'

        u1 :                                y1, d);
        u2 :                                y2, y1);
        u3 :                                y3, y2);
        u4 :                                d, c);
        u5 :                                d, c);
       DELAY:                               o 0 generate
        u6 :                                y4, y3);
       end ge
        u7 :                                 (y2, y1);
     end;
```

This functionality allows you to easily navigate your design for debugging purposes by remembering where you have been, similar to the functionality in most web browsers. The navigation options in the pop-up menu function as follows:

- **Open Instance** — changes your context to the instance you have selected within the source file. This is not available if you have not placed your cursor in, or highlighted the name of, an instance within your source file.

  If any ambiguities exists, most likely due to generate statements, this option opens a dialog box allowing you to choose from all available instances.

- **Ascend Env** — changes your context to the next level up within the design. This is not available if you are at the top-level of your design.

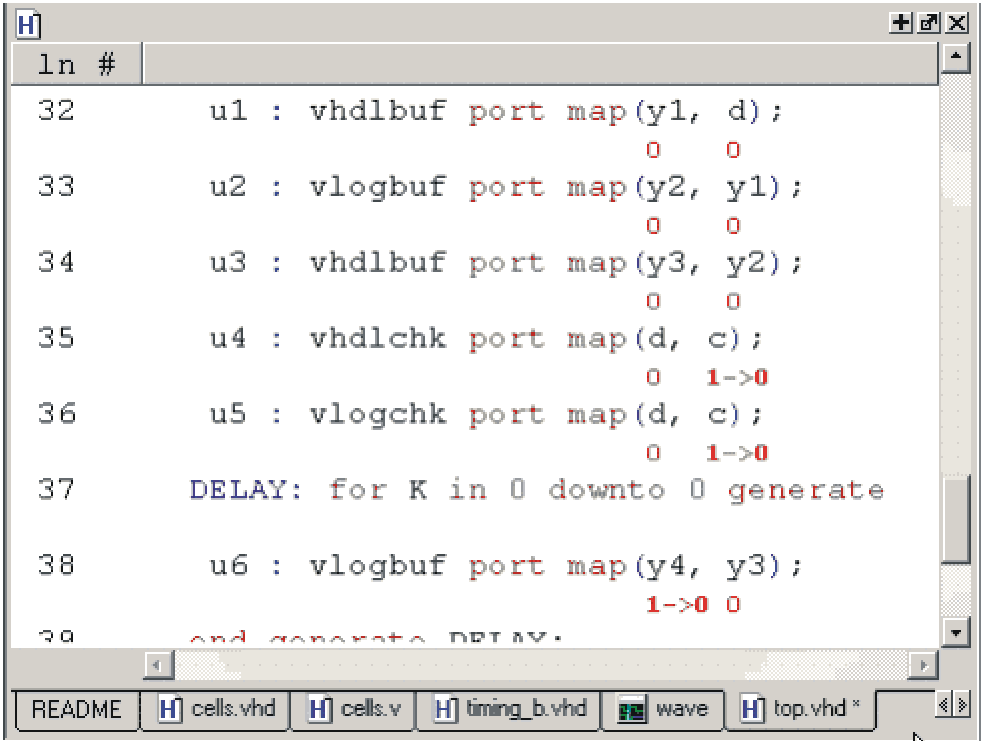
- **Forward/Back** — allows you to change to previously selected contexts. This is not available if you have not changed your context.

The Open Instance option is essentially executing an environment command to change your context, therefore any time you use this command manually at the command prompt, that information is also saved for use with the Forward/Back options.

Debugging with Source Annotation

With source annotation you can interactively debug your design by analyzing your source files in addition to using the Wave and Signal windows. Source annotation displays simulation values, including transitions, for each signal in your source file. Figure 2-3 shows an example of source annotation, where the red values are added below the signals.

Figure 2-3. Source Annotation Example



```
  u1 : vhdlbuf port map(y1, d);
                        0   0
  u2 : vlogbuf port map(y2, y1);
                        0   0
  u3 : vhdlbuf port map(y3, y2);
                        0   0
  u4 : vhdlchk port map(d, c);
                        0  1->0
  u5 : vlogchk port map(d, c);
                        0  1->0
DELAY: for K in 0 downto 0 generate

  u6 : vlogbuf port map(y4, y3);
                        1->0 0
end generate DELAY;
```

Turn on source annotation by selecting **View > Source > Show Source Annotation** or by right-clicking a source file and selecting **Show Source Annotation**. Note that transitions are displayed only for those signals that you have logged.

To analyze the values at a given time of the simulation you can either:

- Show the signal values at the current simulation time. This is the default behavior. The window automatically updates the values as you perform a run or a single-step action.
- Show the signal values at current cursor position in the Wave window.

You can switch between these two settings by performing the following actions:

1. **Tools** > **Edit Preferences** > **By Window** tab
2. select **Source Windows** in the Window List box
3. select **Annotation** in the Category box
4. select Use Values: at Active Waveform cursor or Current simulation time.

You can highlight a specific signal in the Wave window by double-clicking on an annotation value in the source file.

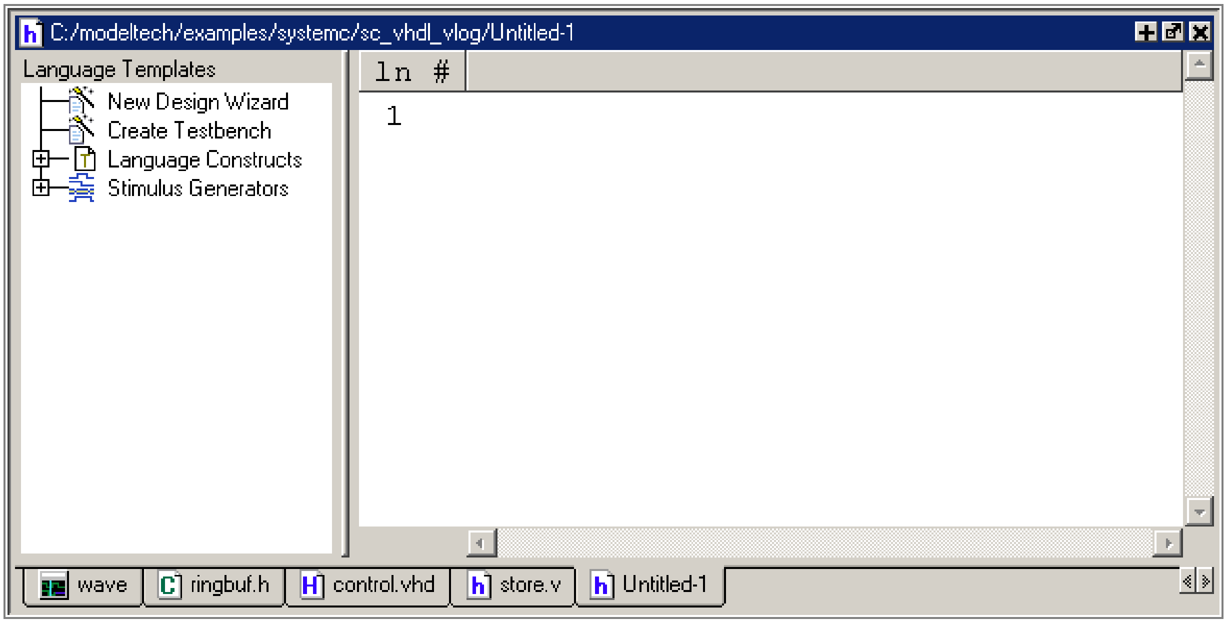
Language Templates

ModelSim language templates help you write code. They are a collection of wizards, menus, and dialogs that produce code for new designs, testbenches, language constructs, logic blocks, etc.

Note

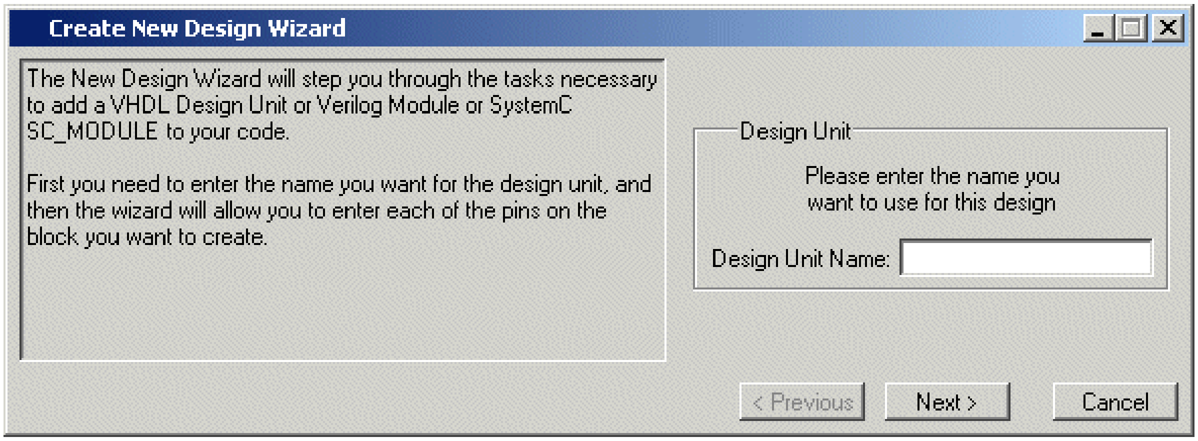
The language templates are not intended to replace thorough knowledge of coding. They are intended as an interactive "reference" for creating small sections of code. If you are unfamiliar with a particular language, you should attend a training class or consult one of the many available books.

To use the templates, either open an existing file, or select **File** > **New** > **Source** to create a new file. Once the file is open, select **View** > **Source** > **Show Language Templates**. This displays a pane that shows the available templates.



The templates that appear depend on the type of file you create. For example Module and Primitive templates are available for Verilog files, and Entity and Architecture templates are available for VHDL files.

Double-click an object in the list to open a wizard or to begin creating code. Some of the objects bring up wizards while others insert code into your source file. The dialog below is part of the wizard for creating a new design. Simply follow the directions in the wizards.



Create New Design Wizard

The New Design Wizard will step you through the tasks necessary to add a VHDL Design Unit or Verilog Module or SystemC SC_MODULE to your code.

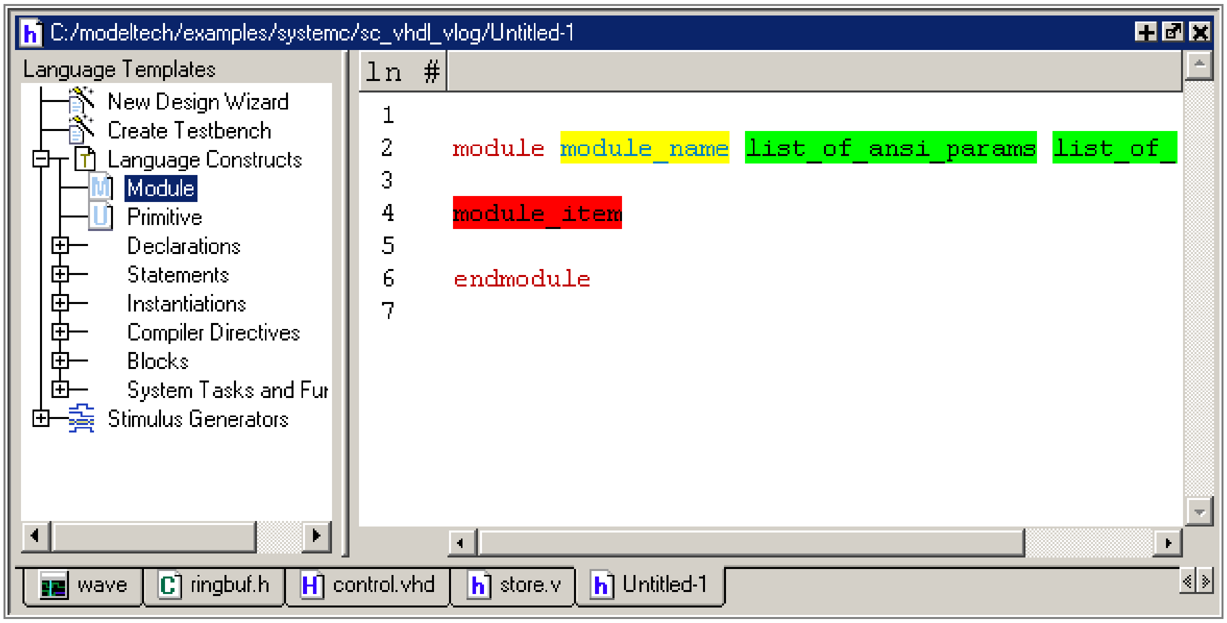
First you need to enter the name you want for the design unit, and then the wizard will allow you to enter each of the pins on the block you want to create.

Design Unit

Please enter the name you want to use for this design

Design Unit Name:

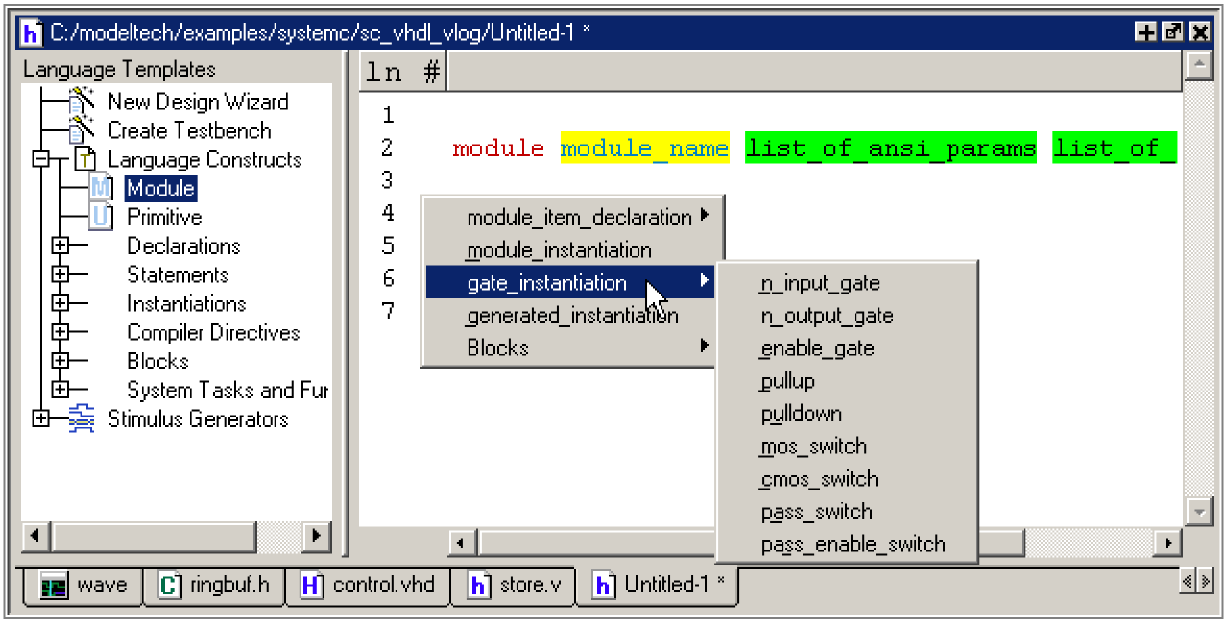
Code inserted into your source contains a variety of highlighted fields. The example below shows a module statement inserted from the Verilog template.



C:/modeltech/examples/systemc/sc_vhdl_vlog/Untitled-1

```
module module_name list_of_ansi_params list_of_

module_item

endmodule
```

Some of the fields, such as *module_name* in the example above, are to be replaced with names you type. Other fields can be expanded by double-clicking and still others offer a context menu of options when double-clicked. The example below shows the menu that appears when you double-click *module_item then select gate_instantiation*.



```
module module_name list_of_ansi_params list_of_
```

Setting File-Line Breakpoints

You can easily set File-line breakpoints in a Source window using your mouse. Click on a red line number at the left side of the Source window, and a red circle denoting a breakpoint will appear. The breakpoints are toggles – click once to create the breakpoint; click again to disable or enable the breakpoint.

Note

When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See Restoring Full Design Visibility by Disabling Auto vopt and Design Object Visibility and the +acc Argument.

To delete the breakpoint completely, right click the red circle, and select **Remove Breakpoint**. Other options on the context menu include:

- **Disable/Enable Breakpoint** — Deactivate or activate the selected breakpoint.
- **Edit Breakpoint** — Open the File Breakpoint dialog to change breakpoint arguments.
- **Edit All Breakpoints** — Open the Modify Breakpoints dialog

Checking Object Values and Descriptions

There are two quick methods to determine the value and description of an object displayed in the Source window:

- select an object, then right-click and select **Examine** or **Describe** from the context menu
- pause over an object with your mouse pointer to see an examine pop-up

Select **Tools > Options > Examine Now** or **Tools > Options > Examine Current Cursor** to choose at what simulation time the object is examined or described.

You can also invoke the examine and/or describe commands on the command line or in a macro.

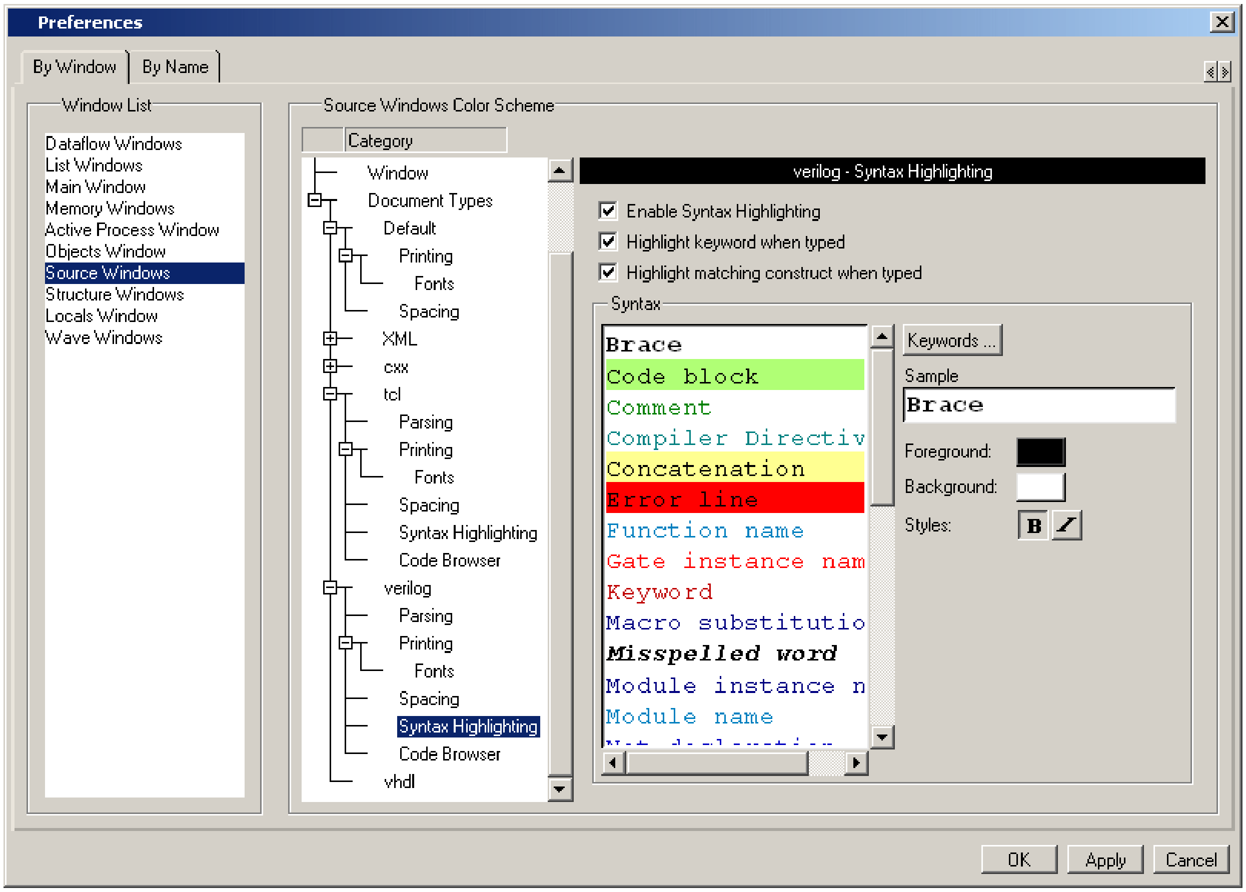
Marking Lines with Bookmarks

Source window bookmarks are blue circles that mark lines in a source file. These graphical icons may ease navigation through a large source file by "highlighting" certain lines.

As noted above in the discussion about finding text in the Source window, you can insert bookmarks on any line containing the text for which you are searching. The other method for inserting bookmarks is to right-click a line number and select **Add/Remove Bookmark**. To remove a bookmark, right-click the line number and select Add/Remove Bookmark again.

Customizing the Source Window

You can customize a variety of settings for Source windows. For example, you can change fonts, spacing, colors, syntax highlighting, and so forth. To customize Source window settings, select **Tools > Edit Preferences**. This opens the Preferences dialog. Select **Source Windows** from the Window List.



☑ Enable Syntax Highlighting

☑ Highlight keyword when typed

☑ Highlight matching construct when typed

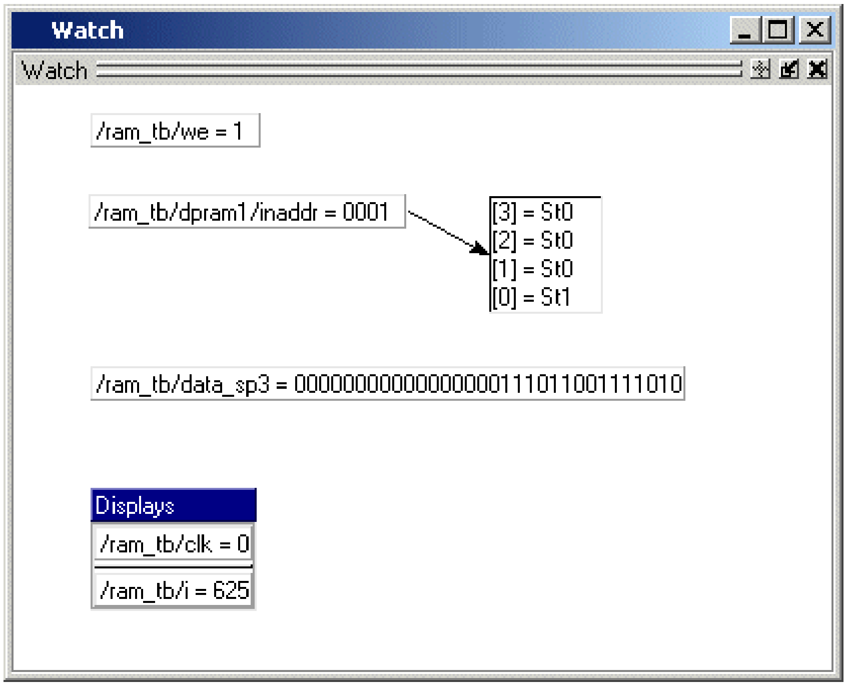
Select an item from the Category list and then edit the available properties on the right. Click OK or Apply to accept the changes.

The changes will be active for the next Source window you open. The changes are saved automatically when you quit ModelSim.

Watch Pane

The Watch pane shows values for signals and variables at the current simulation time. Unlike the Objects or Locals pane, the Watch pane allows you to view any signal or variable in the design regardless of the current context.

.



/ram_tb/we = 1

/ram_tb/dpram1/inaddr = 0001

[3] = St0
[2] = St0
[1] = St0
[0] = St1

/ram_tb/data_sp3 = 00000000000000000111011001111010

/ram_tb/clk = 0

/ram_tb/i = 625

you can view the following objects in the watch pane.

- VHDL objects — signals, aliases, generics, constants, and variables
- Verilog objects — nets, registers, variables, named events, and module parameters
- SystemC objects — primitive channels and ports
- Virtual objects — virtual signals and virtual functions

Adding Objects to the Pane

To add objects to the Watch pane, drag-and-drop objects from the Structure tab, Objects pane, or Locals pane. Alternatively, use the add watch command.

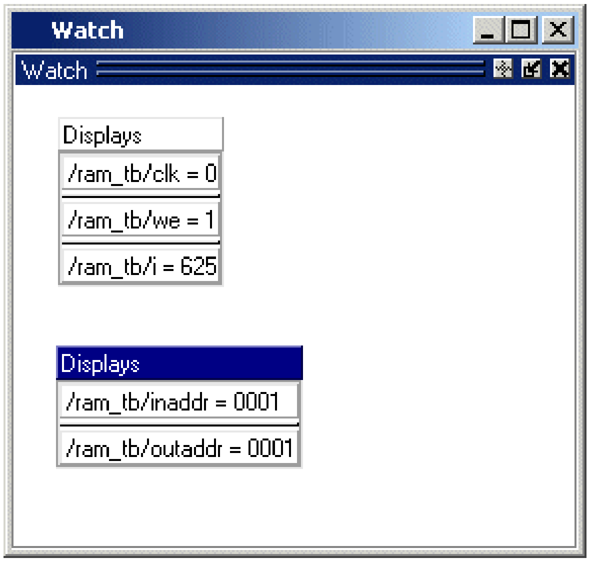
Expanding Objects to Show Individual Bits

If you add an array or record to the Watch pane, you can view individual bit values by double-clicking the array or record. As shown in the graphic above, */ram_tb/dpram1/inaddr* has been expanded to show all the individual bit values. Notice the arrow that "ties" the array to the individual bit display.

Grouping and Ungrouping Objects

You can group objects in the Watch pane so they display and move together. Select the objects, then right click one of the objects and choose Group.

In the graphic below, two different sets of objects have been grouped together.



/ram_tb/clk = 0
/ram_tb/we = 1
/ram_tb/i = 625

/ram_tb/inaddr = 0001
/ram_tb/outaddr = 0001

To ungroup them, right-click the group and select Ungroup.

Saving and Reloading Format Files

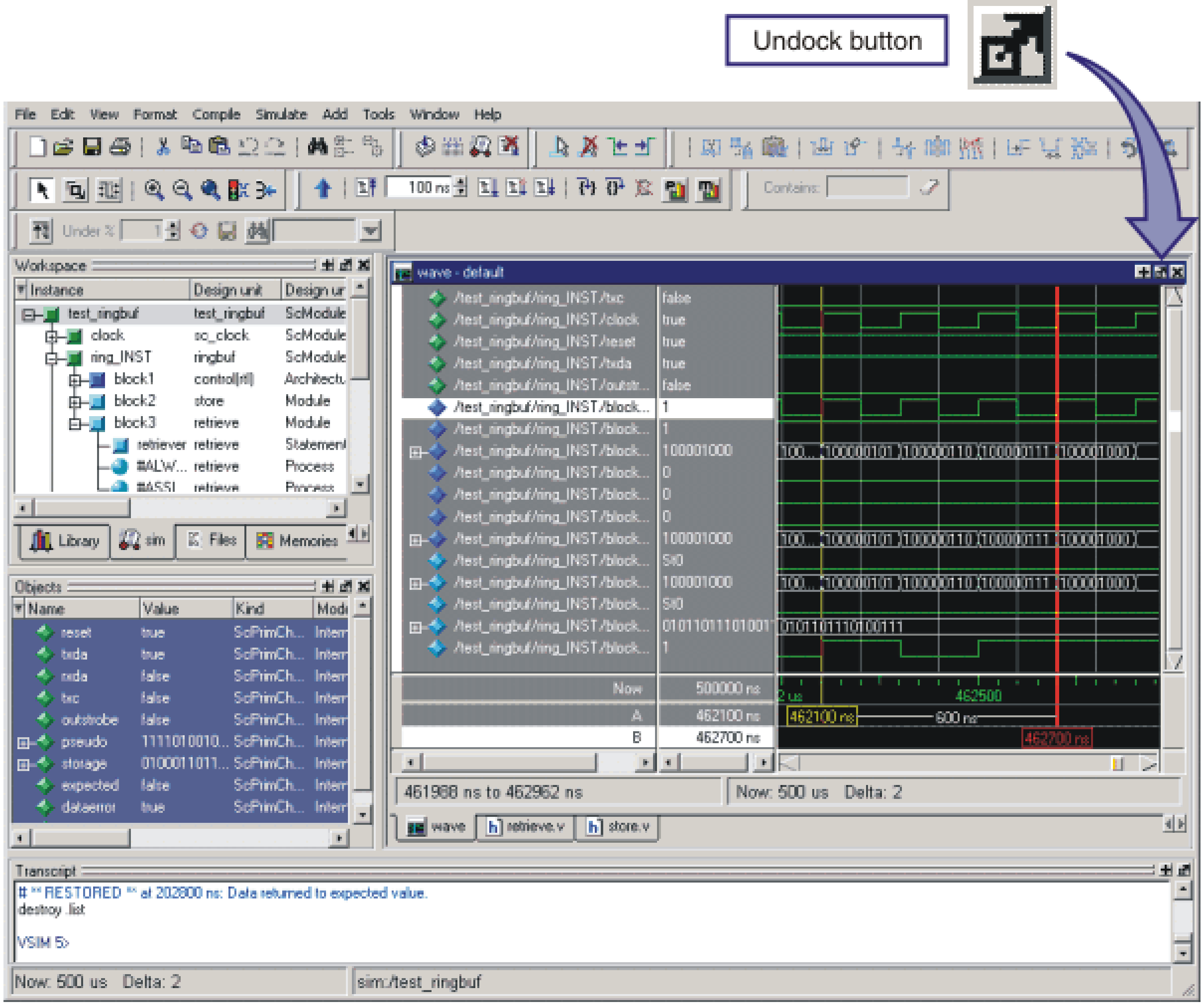
You can save a format file (a DO file, actually) that will redraw the contents of the Watch window. Right-click anywhere in the window and select **Save Format**.

Once you have saved the file, you can reload it by right-clicking and selecting **Load Format**.

Wave Window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as waveforms and their values.

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by pressing the Undock button in the window header or by using the **view -undock wave** command. The preference variable **PrefMain(ViewUnDocked) wave** can be used to control this default behavior. Setting this variable will open the Wave Window undocked each time you start ModelSim.



Instance	Design unit	Design ur
test_ringbuf	test_ringbuf	ScModule
clock	sc_clock	ScModule
ring_INST	ringbuf	ScModule
block1	control(rtl)	Architectu
block2	store	Module
block3	retrieve	Module
retriever	retrieve	Statement
#ALW...	retrieve	Process
#ASSI	retrieve	Process

Name	Value	Kind	Mod
reset	true	ScPrimCh...	Inter
txda	true	ScPrimCh...	Inter
rxda	false	ScPrimCh...	Inter
txc	false	ScPrimCh...	Inter
outstrobe	false	ScPrimCh...	Inter
pseudo	1111010010...	ScPrimCh...	Inter
storage	0100011011...	ScPrimCh...	Inter
expected	false	ScPrimCh...	Inter
dataerror	true	ScPrimCh...	Inter

/test_ringbuf/ring_INST/txc	false
/test_ringbuf/ring_INST/clock	true
/test_ringbuf/ring_INST/reset	true
/test_ringbuf/ring_INST/txda	true
/test_ringbuf/ring_INST/outstr...	false
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	100001000
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	100001000
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	100001000
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	010110111010011
/test_ringbuf/ring_INST/block...	1
Now	500000 ns
A	462100 ns
B	462700 ns

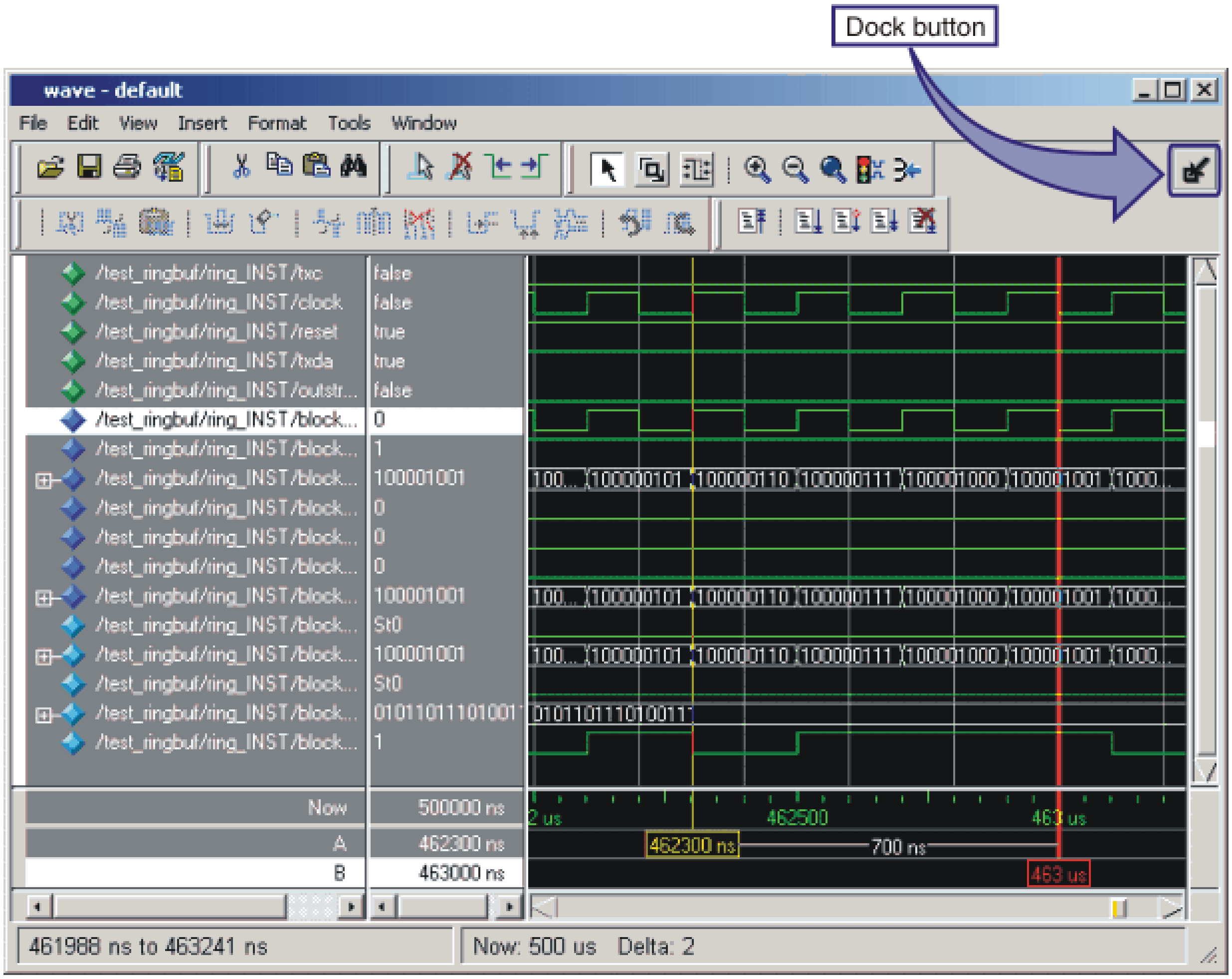
461988 ns to 462962 ns   Now: 500 us   Delta: 2

wave   retrieve.v   store.v

```
# ** RESTORED ** at 202800 ns: Data returned to expected value.
destroy .list

VSIM 5>
```

Now: 500 us   Delta: 2   sim:/test_ringbuf

Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.


Dock button

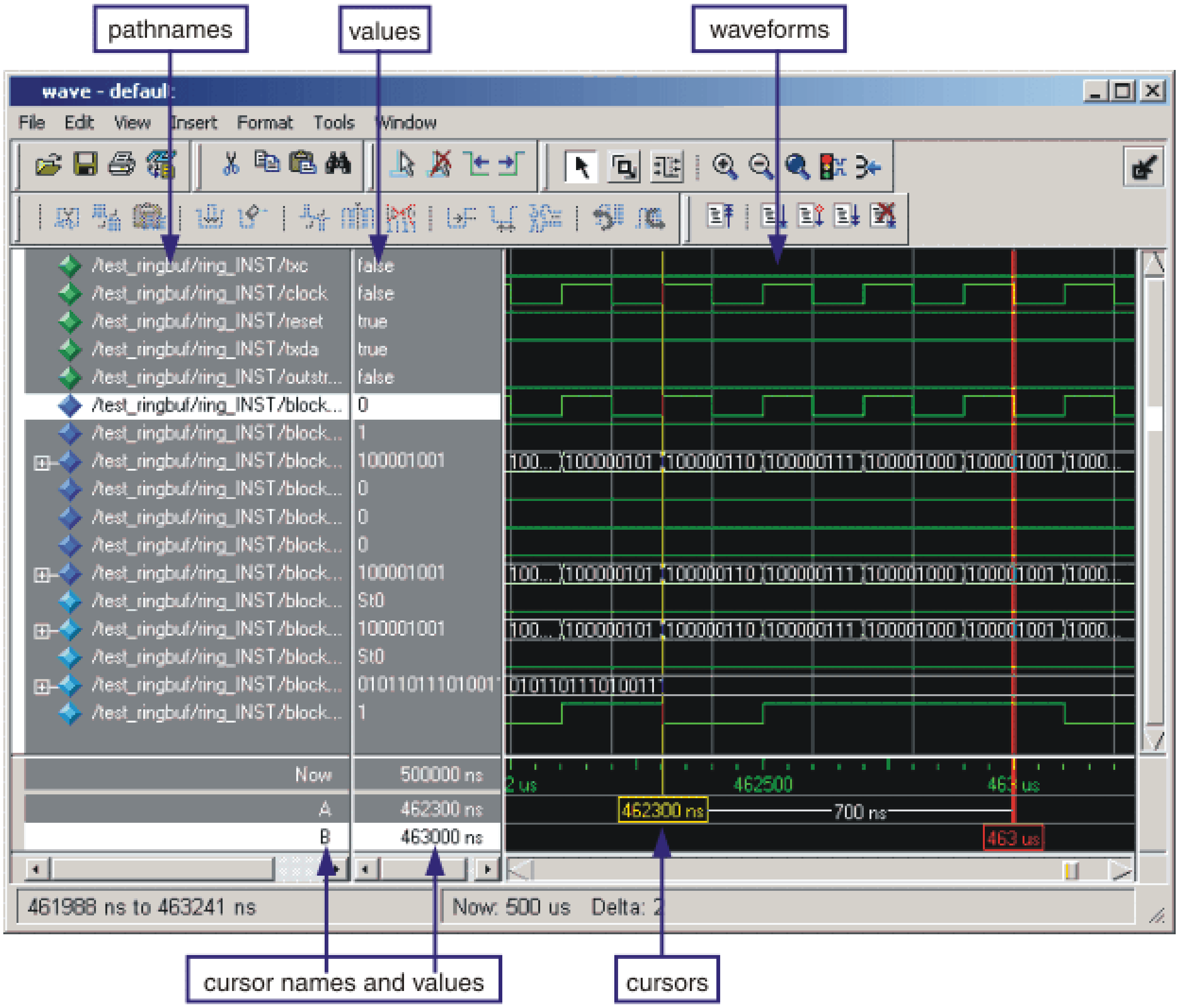
/test_ringbuf/ring_INST/txc	false
/test_ringbuf/ring_INST/clock	false
/test_ringbuf/ring_INST/reset	true
/test_ringbuf/ring_INST/txda	true
/test_ringbuf/ring_INST/outstr...	false
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	010110111010011
/test_ringbuf/ring_INST/block...	1
Now	500000 ns
A	462300 ns
B	463000 ns

461988 ns to 463241 ns

Now: 500 us Delta: 2

If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.


pathnames
values
waveforms
wave - default
File Edit View Insert Format Tools Window
/test_ringbuf/ring_INST/txc
/test_ringbuf/ring_INST/clock
/test_ringbuf/ring_INST/reset
/test_ringbuf/ring_INST/txda
/test_ringbuf/ring_INST/outstr...
/test_ringbuf/ring_INST/block...
false
false
true
true
false
0
1
100001001
St0
Now 500000 ns
A 462300 ns
B 463000 ns
462500
463 us
462300 ns
700 ns
463 us
461988 ns to 463241 ns
Now: 500 us Delta: 2
cursor names and values
cursors

The following types of objects can be viewed in the Wave window

- VHDL objects (indicated by a dark blue diamond) — signals, aliases, process variables, and shared variables
- Verilog objects (indicated by a light blue diamond) — nets, registers, variables, and named events
- SystemC objects
  (indicated by a green diamond) — primitive channels and ports
  (indicated by a green four point star) — transaction streams and their element
- Virtual objects (indicated by an orange diamond) — virtual signals, buses, and functions, see; Virtual Objects for more information

- Comparison objects (indicated by a yellow triangle) — comparison region and comparison signals; see Waveform Compare for more information
- Created waveforms (indicated by a red dot on a diamond) — see Generating Stimulus with Waveform Editor

The data in the object values pane is very similar to the Objects window, except that the values change dynamically whenever a cursor in the waveform pane is moved.

At the bottom of the waveform pane you can see a time line, tick marks, and the time value of each cursor’s position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.

You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu. You can reuse any formatting changes you make by saving a Wave window format file (see Saving the Window Format).

Wave Window Panes

The sections below describe the various Wave window panes.

Pathname Pane

The pathname pane displays signal pathnames. Signals can be displayed with full pathnames, as shown here, or with only the leaf element displayed. You can increase the size of the pane by clicking and dragging on the right border. The selected signal is highlighted.

The white bar along the left margin indicates the selected dataset (see Splitting Wave Window Panes).

Value Pane

The value pane displays the values of the displayed signals.

The radix for each signal can be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII, or default. The default radix can be set by selecting **Simulate** > **Runtime Options**.

Note

When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

The data in this pane is similar to that shown in the Objects Pane, except that the values change dynamically whenever a cursor in the waveform pane is moved.

Waveform Pane

The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors. Signal values can be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. The radix of each signal can be set individually by selecting the signal and then choosing . The default radix is logic.

If you rest your mouse pointer on a signal in the waveform pane, a popup displays with information about the signal. You can toggle this popup on and off in the **Wave Window Properties** dialog.

Dashed signal lines in the waveform pane indicate weak or ambiguous strengths of Verilog states. See Verilog States in the Mixed-Language Simulation chapter.

Cursor Panes

There are three cursor panes–the left pane shows the cursor names; the middle pane shows the current simulation time and the value for each cursor; and the right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors can be displayed. See Measuring Time with Cursors in the Wave Window for more information.

Wave Window Toolbar

The Wave window toolbar (in the undocked Wave window) gives you quick access to these ModelSim commands and functions.

Table 2-12.

Button	Menu equivalent	Other options
📂 **Open Dataset** open a previously saved dataset	File > Open	File > Open from Main window when Transcript window sim tab is active
💾 **Save Format** save the current Wave window display and signal preferences to a DO (macro) file	File > Save	none
🖨 **Print** print a user-selected range of the current Wave window display to a printer or a file	File > Print File > Print Postscript	none

Table 2-12.

Button	Menu equivalent	Other options
Export Waveform export a created waveform	File > Export > Waveform	none
Cut cut the selected signal from the Wave window	Edit > Cut	right mouse in pathname pane > Cut
Copy copy the signal selected in the pathname pane	Edit > Copy	right mouse in pathname pane > Copy
Paste paste the copied signal above another selected signal	Edit > Paste	right mouse in pathname pane > Paste
Find find a name or value in the Wave window	Edit > Find	<control-f> Windows <control-s> UNIX
Insert Cursor add a cursor to the waveform pane	Insert > Cursor	right click in cursor pane and select New Cursor
Delete Cursor delete the selected cursor from the window	Edit > Delete Cursor	right mouse in cursor pane > Delete Cursor n
Find Previous Transition locate the previous signal value change for the selected signal	Edit > Search (Search Reverse)	keyboard: Shift + Tab **left** <arguments> see left command
Find Next Transition locate the next signal value change for the selected signal	Edit > Search (Search Forward)	keyboard: Tab **right** <arguments> see right command
Select Mode set mouse to Select Mode – click left mouse button to select, drag middle mouse button to zoom	View > Zoom > Mouse Mode > Select Mode	none
Zoom Mode set mouse to Zoom Mode – drag left mouse button to zoom, click middle mouse button to select	View > Zoom > Mouse Mode > Zoom Mode	none

Table 2-12.

Button	Menu equivalent	Other options
Zoom In 2x zoom in by a factor of two from the current view	View > Zoom > Zoom In	keyboard: i I or + right mouse in wave pane > Zoom In
Zoom Out 2x zoom out by a factor of two from current view	View > Zoom > Zoom Out	keyboard: o O or - right mouse in wave pane > Zoom Out
Zoom in on Active Cursor center active cursor in the display and zoom in	View > Zoom > Zoom Cursor	keyboard: c or C
Zoom Full zoom out to view the full range of the simulation from time 0 to the current time	View > Zoom > Zoom Full	keyboard: f or F right mouse in wave pane > Zoom Full
Stop Wave Drawing halts any waves currently being drawn in the Wave window	none	.wave.tree interrupt
Show Drivers display driver(s) of the selected signal, net, or register in the Dataflow window	[Dataflow window] Navigate > Expand net to drivers	**[Dataflow window] Expand net to all drivers** right mouse in wave pane > Show Drivers
Restart reloads the design elements and resets the simulation time to zero, with the option of keeping the current formatting, breakpoints, and WLF file	Main menu: Simulate > Run > Restart	restart <arguments>
Run run the current simulation for the default time length	Main menu: Simulate > Run > Run <default_length>	use the run command at the VSIM prompt
Continue Run continue the current simulation run	Main menu: Simulate > Run > Continue	use the run **-continue** command at the VSIM prompt
Run -All run the current simulation forever, or until it hits a breakpoint or specified break event	Main menu: Simulate > Run > Run -All	use the run **-all** command at the VSIM prompt

Table 2-12.

Button	Menu equivalent	Other options
Break stop the current simulation run	none	none
Find First Difference find the first difference in a waveform comparison	none	none
Find Previous Annotated Difference find the previous annotated difference in a waveform comparison	none	none
Find Previous Difference find the previous difference in a waveform comparison	none	none
Find Next Difference find the next difference in a waveform comparison	none	none
Find Next Annotated Difference find the next annotated difference in a waveform comparison	none	none
Find Last Difference find the last difference in a waveform comparison	none	none

Waveform Editor Toolbar

ModelSim's waveform editor has its own toolbar. The toolbar becomes active once you add an editable wave to the Wave window. Refer to Generating Stimulus with Waveform Editor for more details.

Table 2-13.

Button	Menu equivalent	Other options
Cut Wave cut the selected section of the waveform to the clipboard	Edit > Edit Wave > Cut	wave edit **cut**
Copy Wave copy the selected section of the waveform to the clipboard	Edit > Edit Wave > Copy	**wave edit copy**

Table 2-13.

Button	Menu equivalent	Other options
Paste Wave paste the wave from the clipboard	Edit > Edit Wave > Paste	**wave edit paste**
Insert Pulse Insert a transition at the selected time	Edit > Edit Wave > Insert Pulse	**wave edit insert_pulse**
Delete Edge Delete the selected transition	Edit > Edit Wave > Delete Edge	**wave edit delete**
Invert Invert the selected section of the waveform	Edit > Edit Wave > Invert	**wave edit invert**
Mirror Mirror the selected section of the waveform	Edit > Edit Wave > Mirror	**wave edit mirror**
Change Value Change the value of the selected section of the waveform	Edit > Edit Wave > Value	**wave edit change_value**
Stretch Edge Move the selected edge by increasing/decreasing waveform duration	Edit > Edit Wave > Stretch Edge	**wave edit stretch**
Move Edge Move the selected edge without increasing/decreasing waveform duration	Edit > Edit Wave > Move Edge	**wave edit move**
Extend All Waves Increase the duration of all editable waves	Edit > Edit Wave > Extend All Waves	**wave edit extend**
Wave Undo Undo a previous waveform edit	Edit > Edit Wave > Undo	**wave edit undo**
Wave Redo Redo a previously undone waveform edit	Edit > Edit Wave > Redo	**wave edit redo**

Chapter 3 Optimizing Designs with vopt

Before beginning the task of simulating your design using the ModelSim tool, you must make a decision to run simulation with or without the built-in tool optimizations. These optimizations are performed automatically by default as of version 6.2 of the tool, and are designed to maximize simulator performance. In some designs, these optimizations can yield performance improvements over non-optimized runs. However, the visibility of design objects may impact your results because of those optimizations, so it is important to make an informed decision as to how best to apply optimizations to your design.

The command that performs global optimizations in ModelSim is called vopt. This chapter discusses the **vopt** functionality, the effects of optimization on your design, and how to customize the application of **vopt** to your design. For details on command syntax and usage, please refer to vopt in the Reference Manual.

Before discussing **vopt** specifics, let us briefly discuss the opposite ends of the spectrum of optimization available to you with the ModelSim tool, and their inherent trade-offs.

Optimization Trade-Offs

ModelSim loads and simulates designs fully optimized by default, however you can choose to disable the optimizations. The following table summarizes the main differences between running optimized vs. non-optimized simulations, characterizing the two opposite ends of the spectrum.

Table 3-1. Optimized vs. Non-Optimized Mode of Simulation

	Purpose and characteristics
Optimized / Limited Visibility	Default operation. **vopt** is run automatically, globally on the design. • Used for regression testing and for large (or static) designs • Faster simulation than non-optimized • Visibility into modules is limited (signals and processes may be optimized away)
Non-Optimized / Increased Visibility	Accomplished by not running **vopt** • Used for quick compile/simulate turnaround for small (or dynamic) designs • Complete design visibility and debugability • Slower simulation than optimized mode

Running with Default Optimization

By default, ModelSim optimizations (through **vopt**) are performed on all designs. In some cases when running in this mode, the global optimizations may have an impact on the visibility of your design results. You may expect to see certain signals and processes which do not appear to exist. If this is the case, you need to customize the optimization of your design by removing the optimizations from those modules for which the data is missing.

Note

The default optimization behavior of **vopt** may differ from what is documented in this chapter if you are using any *modelsim.ini* file other than the one shipped with the tool. Refer to "Customized Optimization Options" for information on possible optimization settings and "VoptFlow" for information on *modelsim.ini* file settings.

Customized Optimization Options

You can run your design in ModelSim with as much, or as little, design optimization as you prefer, using one of several approaches. You can:

- Accept the default optimizations:

  Run with full optimization, and limited visibility. Refer to "Usage Flow with Automatic vopt" for more information.

- Preserving visibility during optimization:

  Preserve visibility for all top-level modules

  ```
  vopt mydesign +acc -o mydesign_opt       //explicit optimization using vopt command
  vsim mydesign -voptargs="+acc"            //implicit optimizatin using vsim command
  ```

  Preserve visibility for a specific module using the **+acc** argument to vopt. For example:

  ```
  vopt top +acc+mod1                           // preserve visibility for module "mod1"
  ```

  Preserve visibility when optimization is performed implicitly with vsim:

  ```
  vsim top -voptargs"+acc=rnp+dut"
                         //preserve visibility for ports, nets, registers in "dut"
  ```

  Refer to "Design Object Visibility and the +acc Argument" for more details on using **+acc**.

- Customize your application of **vopt** to the design:

  Disable automatic optimizations by setting **VoptFlow** = **0** in the *modelsim.ini* file. Then explicitly use the **vopt** command on a module by module basis. Refer to the **vopt** command for details on command syntax and arguments. When compiling with vcom or vlog, you need to apply the **-vopt** argument (Refer to "Manual Optimization and -vopt Argument").

Restoring Full Design Visibility by Disabling Auto vopt

Disable the automatic optimization of your design by setting the value of the **VoptFlow** variable to “0” in the *modelsim.ini* file:

```
VoptFlow = 0
```

Manual Optimization and -vopt Argument

After turning off the optimizations for the whole design, you can manually run **vopt** on specified modules of your design. When you do, use the **-vopt** argument to vcom / vlog to compile the design. This argument instructs the compiler that **vopt** is being run on the design, and to skip the code production that normally occurs during compilation. Otherwise, without the **-vopt** argument, **vcom/vlog** produces code, and then **vopt** reproduces the same code in an optimized format. The initial code production by **vcom/vlog** is unnecessary.

Usage Flow with Automatic vopt

This section details how the use flow and how the tool operates with the automatic optimizations enabled.

The **vopt** command loads compiled design units from their libraries and regenerates optimized code, as described in the following usage flow:

1. Compile the design. The vcom / vlog commands compile all your modules with **-vopt** argument (see "Manual Optimization and -vopt Argument").
2. Load the design. When invoked, **vsim** performs the following:

    a. Runs **vopt** in the background the first time it loads the design. The optimized design is named by the tool if no name is specified with the **vopt** -o argument (see Naming the Optimized Design for more details).

    You can pass arguments to **vopt** using the **-voptargs** argument to vsim. For example,

    ```
    vsim -voptargs="+acc=rn"
    ```

    b. Runs **vsim** on the optimized design unit

3. Simulate the design using the run command or GUI simulate commands.

Naming the Optimized Design

You can provide a name for the optimized design using the **-o** argument to **vopt**:

```
% vopt testbench -o opt1
```

Make sure filenames do not use capital letters or any character that is illegal for your platform (e.g., on Windows you cannot use "\").

If you do not specify a name for an optimized design, the design is considered "unnamed" and when vsim is run, the tool assigns a default name of "_opt[number]" to the optimized version of the design. By default, the maximum number of "unnamed" designs ("_opt[number]) is set to 3. To control the number of unnamed designs, use the **-unnamed_designs** argument to **vlib**. See vlib for more details.

Incremental Compilation of Named Designs

The default operation of **vopt -o <name>** is incremental compilation: The tool reuses elements of the design that have not changed. This results in a boost of **vopt** performance when a design has been minimally modified.

Viewing the Optimized Design in the GUI

You can view optimized designs in the GUI, or at the command-line with vdir, and delete them with vdel, etc. For example, a vdir command shows something like the following:

```
OPTIMIZED DESIGN opt1
```

Breakpointing Unavailable with vopt

When running in full optimization mode, breakpoints can not be set. You need to run the design in non-optimized mode (or set +acc arguments) to be able to set breakpoints in the design. See Restoring Full Design Visibility by Disabling Auto vopt and Design Object Visibility and the +acc Argument for more information.

Controlling Optimization from the GUI

Optimization (**vopt**) in the GUI is controlled from the **Simulate** > **Design Optimization** dialog box.

To restore total design visibility from within the GUI:

1. Select **Simulate** > **Design Optimization** > Visibility tab
2. Select "Apply full visibility to all modules (full debug mode)"
3. Select Design tab and select the top-level design unit to simulate
4. Specify an Output Design Name.
5. Select Start Immediately and then click OK.

Optimization Considerations for Verilog Designs

The optimization considerations for Verilog designs include:

- Design Object Visibility and the +acc Argument
- Reporting on Gate-Level Optimizations
- Using Pre-Compiled Libraries
- Event Order and Optimized Designs
- Timing Checks in Optimized Designs

Design Object Visibility and the +acc Argument

Some of the optimizations performed by **vopt** impact design object visibility. For example:

- Many objects are unavailable by name in user interface commands and in the various graphic interface windows.
- Many objects do not have PLI Access handles, potentially affecting the operation of PLI applications. However, a handle is guaranteed to exist for any object that is an argument to a system task or function.
- Many objects will not appear in WLF or VCD files.

In the early stages of design, you may use one or more **+acc** arguments in conjunction with **vopt** to enable access to specific design objects. See the vopt command in the Reference Manual for specific syntax of the +acc argument.

Keep in mind that enabling design object access may reduce simulation performance.

Automatic +acc — Designs with Automatic Optimization

By default, if your design contains any PLI, and the automatic vopt flow is enabled, **vsim** automatically adds a **+acc** to the sub-invocation of **vopt**, which disables most optimizations.

If you want to override the automatic disabling of the optimizations for modules containing PLI, specify the **-no_autoacc** argument to **vsim**.

Manual +acc — Designs without Automatic Optimizations

If you are running without the default vopt optimizations, and your design uses PLI applications that look for object handles in the design hierarchy, then it is likely that you will need to use the **+acc** option. For example, the built-in **$dumpvars** system task is an internal PLI application that requires handles to nets and registers so that it can call the PLI routine **acc_vcl_add()** to monitor changes and dump the values to a VCD file. This requires that access is enabled for the nets and registers on which it operates.

Suppose you want to dump all nets and registers in the entire design, and that you have the following $dumpvars call in your testbench (no arguments to $dumpvars means to dump everything in the entire design):

```
initial $dumpvars;
```

Then you need to optimize your design as follows to enable net and register access for all modules in the design:

```
% vopt +acc=rn testbench
```

As another example, suppose you only need to dump nets (n) and registers (r) of a particular instance in the design (the first argument of **1** means to dump just the variables in the instance specified by the second argument):

```
initial $dumpvars(1, testbench.u1);
```

Then you need to optimize your design as follows (assuming *testbench.u1* refers to the module *design*):

```
% vopt +acc=rn+design testbench
```

Finally, suppose you need to dump everything in the children instances of *testbench.u1* (the first argument of **0** means to also include all children of the instance):

```
initial $dumpvars(0, testbench.u1);
```

Then you need to optimize your design as follows:

```
% vopt +acc=rn+design testbench
```

To gain maximum performance, it may be necessary to enable the minimum required access within the design.

Reporting on Gate-Level Optimizations

You can use the write cell_report and the **-debugCellOpt** argument to the vopt command to obtain information about which cells have and have not been optimized. **write cell_report** produces a text file that lists all modules. Modules with "(cell)" following their names are optimized cells. For example,

```
Module: top
Architecture: fast

Module: bottom (cell)
Architecture: fast
```

In this case, top was not optimized and bottom was.

Using Pre-Compiled Libraries

If the source code is unavailable for any of the modules referenced in a design, then you must search libraries for the precompiled modules using the **-L** or **-Lf** arguments to vopt. The **vopt** command optimizes pre-compiled modules the same as if the source code is available. The optimized code for a pre-compiled module is written to the default ‘work’ library.

The **vopt** command automatically searches libraries specified in the `uselib directive (see Verilog-XL uselib Compiler Directive). If your design uses `uselib directives exclusively to reference modules in other libraries, then you do not need to specify library search arguments.

Note

If you use **-L** or **-Lf** with **vopt**, you must also you use them with vsim when you simulate the design.

Event Order and Optimized Designs

The Verilog language does not require that the simulator execute simultaneous events in any particular order. Optimizations performed by **vopt** may expose event order dependencies that cause a design to behave differently than when run unoptimized. Event order dependencies are considered errors and should be corrected (see Event Ordering in Verilog Designs for details).

Timing Checks in Optimized Designs

Timing checks are performed whether you optimize the design or not. In general, you'll see the same results in either case. However, in a cell where there are both interconnect delays and conditional timing checks, you might see different timing check results.

- Without **vopt: T**he conditional checks are evaluated with non-delayed values, complying with the original IEEE Std 1364-1995 specification.
- With **vopt:** the conditional checks will be evaluated with delayed values, complying with the new IEEE Std 1364-2005 specification.

Chapter 4
Projects

Projects simplify the process of compiling and simulating a design and are a great tool for getting started with ModelSim.

What are Projects?

Projects are collection entities for designs under specification or test. At a minimum, projects have a root directory, a work library, and "metadata" which are stored in a *.mpf* file located in a project's root directory. The metadata include compiler switch settings, compile order, and file mappings. Projects may also include:

- Source files or references to source files
- other files such as READMEs or other project documentation
- local libraries
- references to global libraries
- Simulation Configurations (see Creating a Simulation Configuration)
- Folders (see Organizing Projects with Folders)

Note

Project metadata are updated and stored *only* for actions taken within the project itself. For example, if you have a file in a project, and you compile that file from the command line rather than using the project menu commands, the project will not update to reflect any new compile settings.

What are the Benefits of Projects?

Projects offer benefits to both new and advanced users. Projects

- simplify interaction with ModelSim; you don’t need to understand the intricacies of compiler switches and library mappings
- eliminate the need to remember a conceptual model of the design; the compile order is maintained for you in the project. Compile order is maintained for HDL-only designs.
- remove the necessity to re-establish compiler switches and settings at each session; these are stored in the project metadata as are mappings to source files

- allow users to share libraries without copying files to a local directory; you can establish references to source files that are stored remotely or locally
- allow you to change individual parameters across multiple files; in previous versions you could only set parameters one file at a time
- enable "what-if" analysis; you can copy a project, manipulate the settings, and rerun it to observe the new results
- reload the initial settings from the project *.mpf* file every time the project is opened

Project Conversion Between Versions

Projects are generally not backwards compatible for either number or letter releases. When you open a project created in an earlier version, you will see a message warning that the project will be converted to the newer version. You have the option of continuing with the conversion or cancelling the operation.

As stated in the warning message, a backup of the original project is created before the conversion occurs. The backup file is named *<project name>.mpf.bak* and is created in the same directory in which the original project is located.

Getting Started with Projects

This section describes the four basic steps to working with a project.

- Step 1 — Creating a New Project

  This creates a .mpf file and a working library.

- Step 2 — Adding Items to the Project

  Projects can reference or include source files, folders for organization, simulations, and any other files you want to associate with the project. You can copy files into the project directory or simply create mappings to files in other locations.

- Step 3 — Compiling the Files

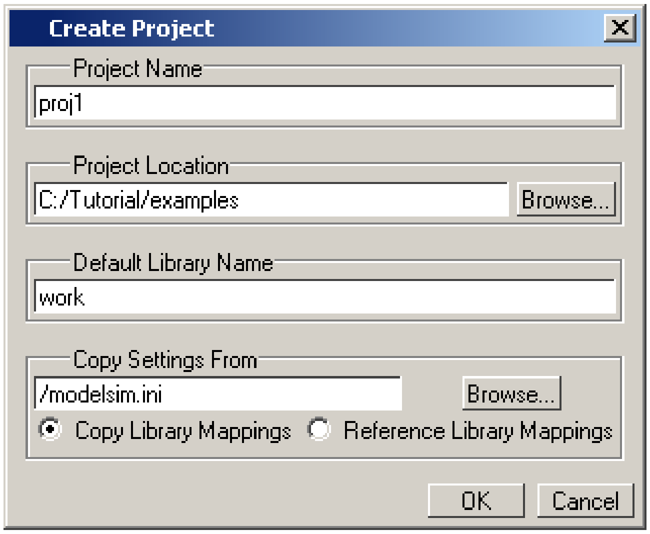
  This checks syntax and semantics and creates the pseudo machine code ModelSim uses for simulation.

- Step 4 — Simulating a Design

  This specifies the design unit you want to simulate and opens a structure tab in the Workspace pane.

Step 1 — Creating a New Project

Select **File** > **New** > **Project** to create a new project. This opens the **Create Project** dialog where you can specify a project name, location, and default library name. You can generally leave the **Default Library Name** set to "work." The name you specify will be used to create a working library subdirectory within the Project Location. This dialog also allows you to reference library settings from a selected .ini file or copy them directly into the project.



Create Project

Project Name: proj1

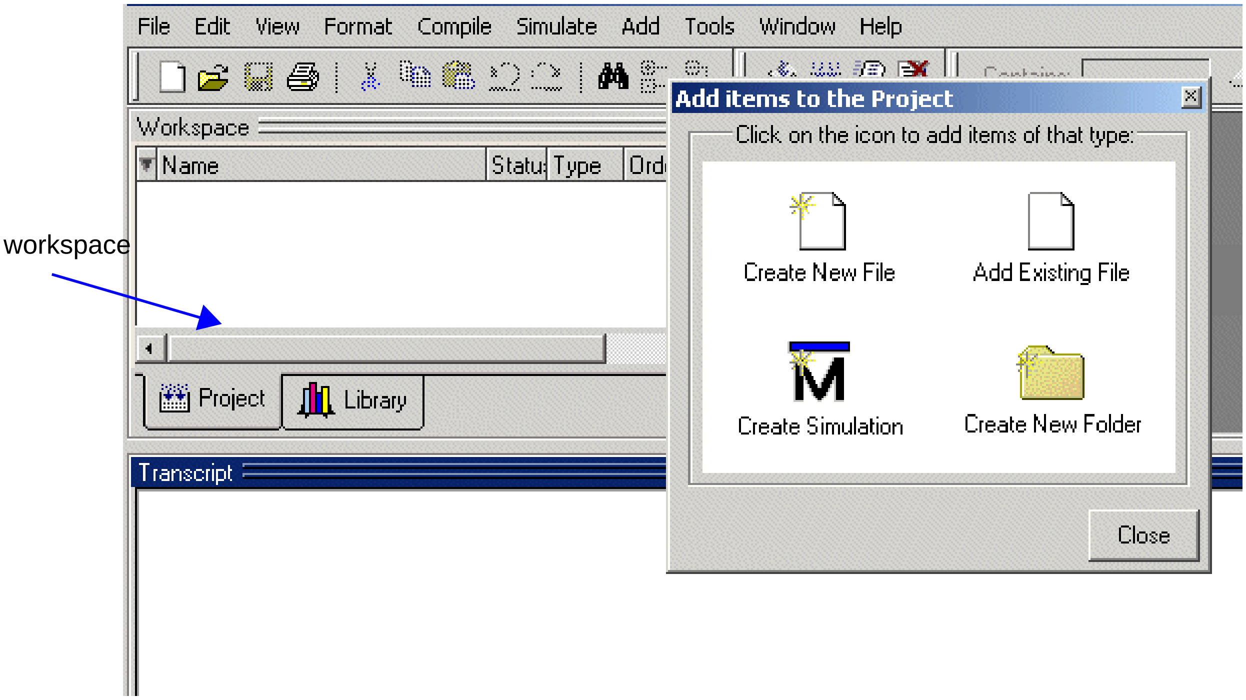
Project Location: C:/Tutorial/examples

Default Library Name: work

Copy Settings From: /modelsim.ini

● Copy Library Mappings ○ Reference Library Mappings

After selecting OK, you will see a blank Project tab in the Workspace pane of the Main window and the **Add Items to the Project** dialog.



The name of the current project is shown at the bottom left corner of the Main window.

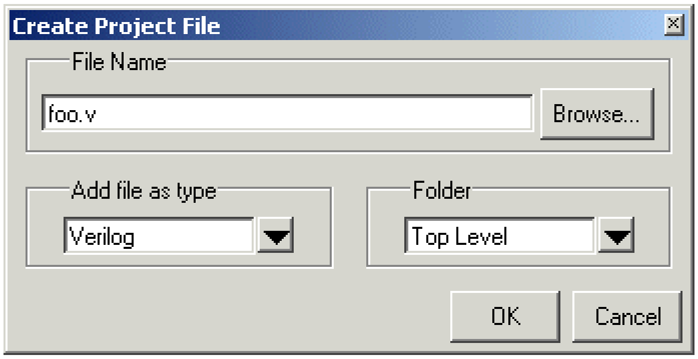
Step 2 — Adding Items to the Project

The **Add Items to the Project** dialog includes these options:

- **Create New File** — Create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor. See below for details.
- **Add Existing File** — Add an existing file. See below for details.
- **Create Simulation** — Create a Simulation Configuration that specifies source files and simulator options. See Creating a Simulation Configuration for details.
- **Create New Folder** — Create an organization folder. See Organizing Projects with Folders for details.

Create New File

The **Create New File** command lets you create a new VHDL, Verilog, SystemC, Tcl, or text file using the Source editor. You can also access this command by selecting **File** > **Add to Project** > **New File** or right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project** > **New File**.



File Name: foo.v

Add file as type: Verilog

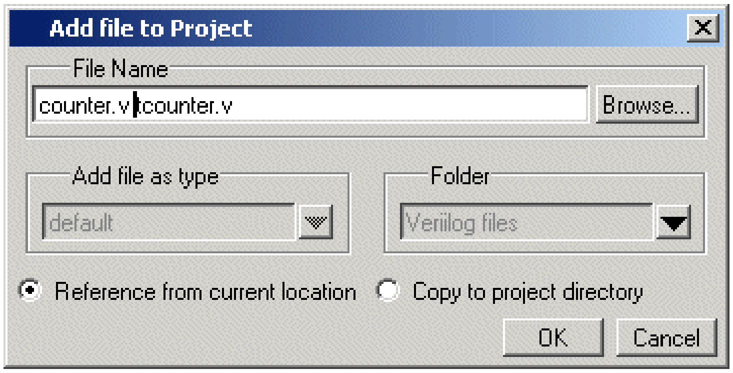
Folder: Top Level

Specify a name, file type, and folder location for the new file.

When you select OK, the file is listed in the Project tab.

Add Existing File

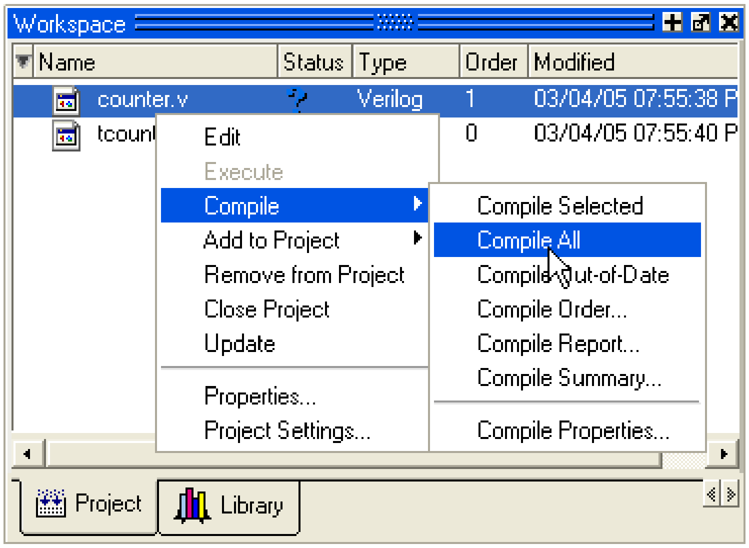
You can also access this command by selecting **File** > **Add to Project** > **Existing File** or by right-clicking (2nd button in Windows; 3rd button in UNIX) in the Project tab and selecting **Add to Project** > **Existing File**.



When you select OK, the file(s) is added to the Project tab.

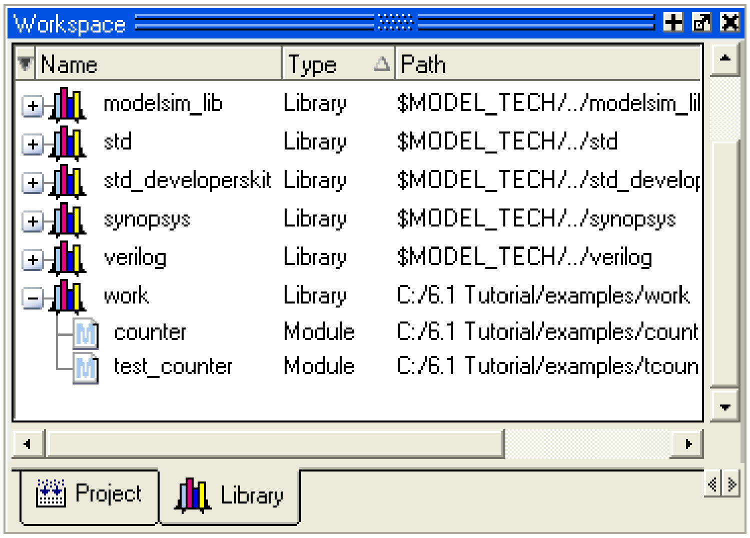
Step 3 — Compiling the Files

The question marks in the Status column in the Project tab denote either the files haven't been compiled into the project or the source has changed since the last compile. To compile the files, select **Compile** > **Compile All** or right click in the Project tab and select **Compile** > **Compile All**.



Name	Status	Type	Order	Modified
counter.v	?	Verilog	1	03/04/05 07:55:38 P
tcoun			0	03/04/05 07:55:40 P

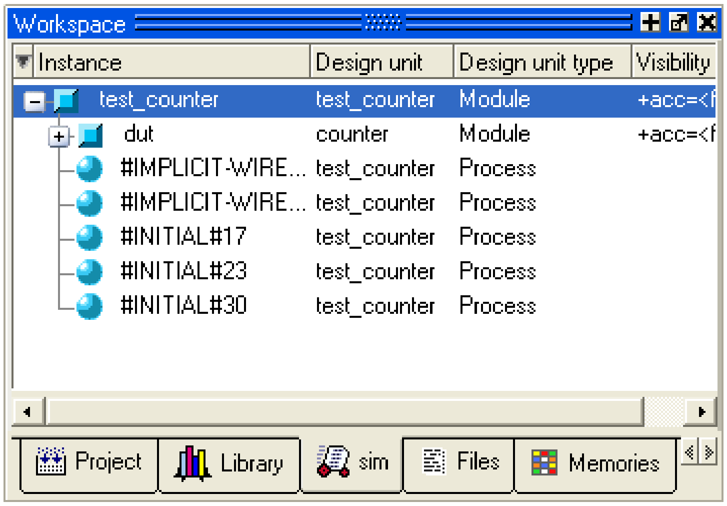
Once compilation is finished, click the Library tab, expand library *work* by clicking the "+", and you will see the compiled design units.



Name	Type	Path
modelsim_lib	Library	$MODEL_TECH/../modelsim_lil
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_develop
synopsys	Library	$MODEL_TECH/../synopsys
verilog	Library	$MODEL_TECH/../verilog
work	Library	C:/6.1 Tutorial/examples/work
counter	Module	C:/6.1 Tutorial/examples/count
test_counter	Module	C:/6.1 Tutorial/examples/tcoun

Step 4 — Simulating a Design

To simulate one of the designs, either double-click the name or right-click the name and select **Simulate**. A new tab named *sim* appears that shows the structure of the active simulation.



Instance	Design unit	Design unit type	Visibility
test_counter	test_counter	Module	+acc=<f
dut	counter	Module	+acc=<f
#IMPLICIT-WIRE...	test_counter	Process	
#IMPLICIT-WIRE...	test_counter	Process	
#INITIAL#17	test_counter	Process	
#INITIAL#23	test_counter	Process	
#INITIAL#30	test_counter	Process	

At this point you are ready to run the simulation and analyze your results. You often do this by adding signals to the Wave window and running the simulation for a given period of time. See the *ModelSim Tutorial* for examples.

Other Basic Project Operations

Open an Existing Project

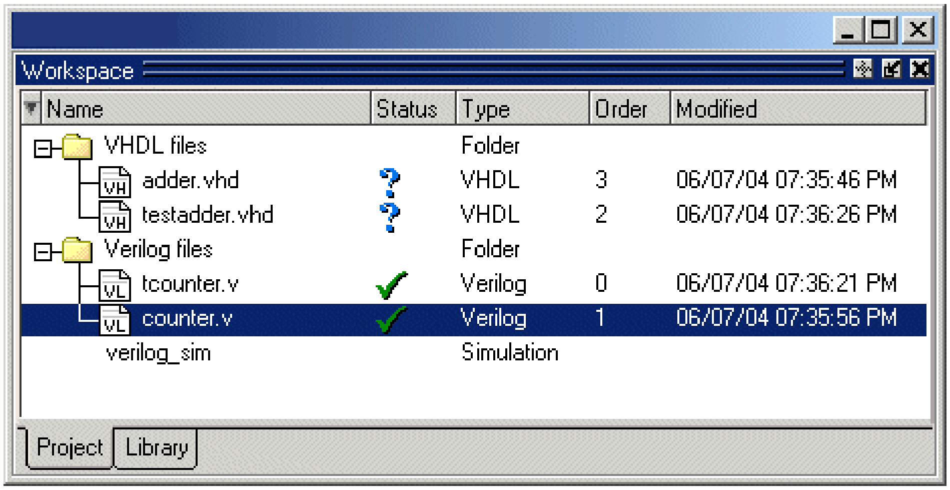
If you previously exited ModelSim with a project open, ModelSim automatically will open that same project upon startup. You can open a different project by selecting **File** > **Open** and choosing Project Files from the Files of type drop-down.

Close a Project

Select **File** > **Close** > **Project** or right-click in the Project tab and select **Close Project**. This closes the Project tab but leaves the Library tab open in the workspace. Note that you cannot close a project while a simulation is in progress.

The Project Tab

The Project tab contains information about the objects in your project. By default the tab is divided into five columns.



Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/04 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/04 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/04 07:36:21 PM
counter.v	✓	Verilog	1	06/07/04 07:35:56 PM
verilog_sim		Simulation		

- **Name** – The name of a file or object.
- **Status** – Identifies whether a source file has been successfully compiled. Applies only to VHDL or Verilog files. A question mark means the file hasn't been compiled or the source file has changed since the last successful compile; an X means the compile failed; a check mark means the compile succeeded; a checkmark with a yellow triangle behind it means the file compiled but there were warnings generated.
- **Type** – The file type as determined by registered file types on Windows or the type you specify when you add the file to the project.
- **Order** – The order in which the file will be compiled when you execute a Compile All command.

- **Modified** – The date and time of the last modification to the file.

You can hide or show columns by right-clicking on a column title and selecting or deselecting entries.

Sorting the List

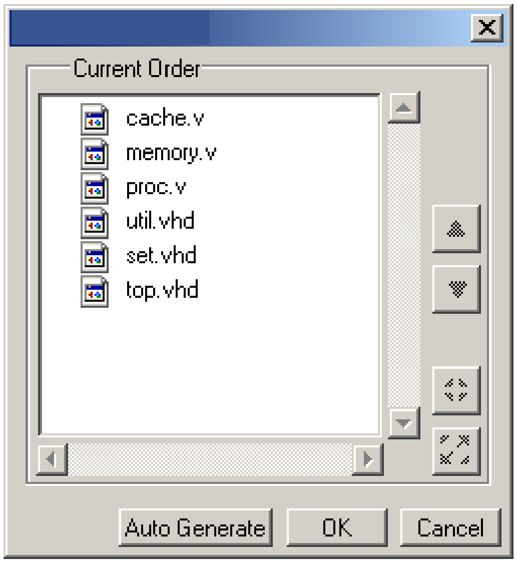
You can sort the list by any of the five columns. Click on a column heading to sort by that column; click the heading again to invert the sort order. An arrow in the column heading indicates which field the list is sorted by and whether the sort order is descending (down arrow) or ascending (up arrow).

Changing Compile Order

The Compile Order dialog box is functional for HDL-only designs. When you compile all files in a project, ModelSim by default compiles the files in the order in which they were added to the project. You have two alternatives for changing the default compile order: 1) select and compile each file individually; 2) specify a custom compile order.

To specify a custom compile order, follow these steps:

1. Select **Compile > Compile Order** or select it from the context menu in the Project tab.



cache.v
memory.v
proc.v
util.vhd
set.vhd
top.vhd

2. Drag the files into the correct order or use the up and down arrow buttons. Note that you can select multiple files and drag them simultaneously.

Auto-Generating Compile Order

Auto Generate is supported for HDL-only designs. The **Auto Generate** button in the Compile Order dialog (see above) "determines" the correct compile order by making multiple passes over the files. It starts compiling from the top; if a file fails to compile due to dependencies, it moves that file to the bottom and then recompiles it after compiling the rest of the files. It continues in this manner until all files compile successfully or until a file(s) can't be compiled for reasons other than dependency.

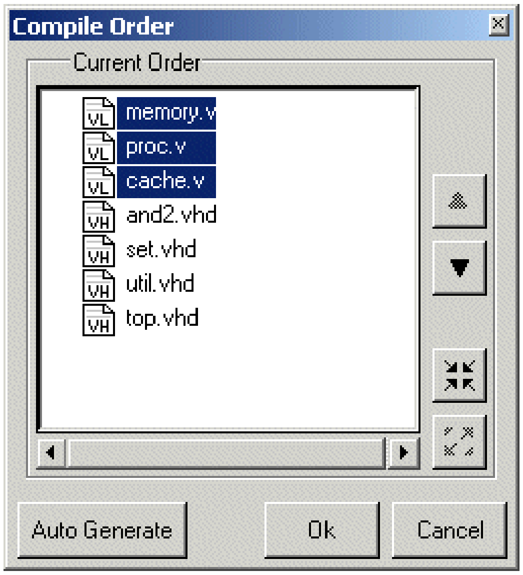
Files can be displayed in the Project tab in alphabetical or compile order (by clicking the column headings). Keep in mind that the order you see in the Project tab is not necessarily the order in which the files will be compiled.

Grouping Files

You can group two or more files in the Compile Order dialog so they are sent to the compiler at the same time. For example, you might have one file with a bunch of Verilog define statements and a second file that is a Verilog module. You would want to compile these two files together.

To group files, follow these steps:

1. Select the files you want to group.



2. Click the Group button.

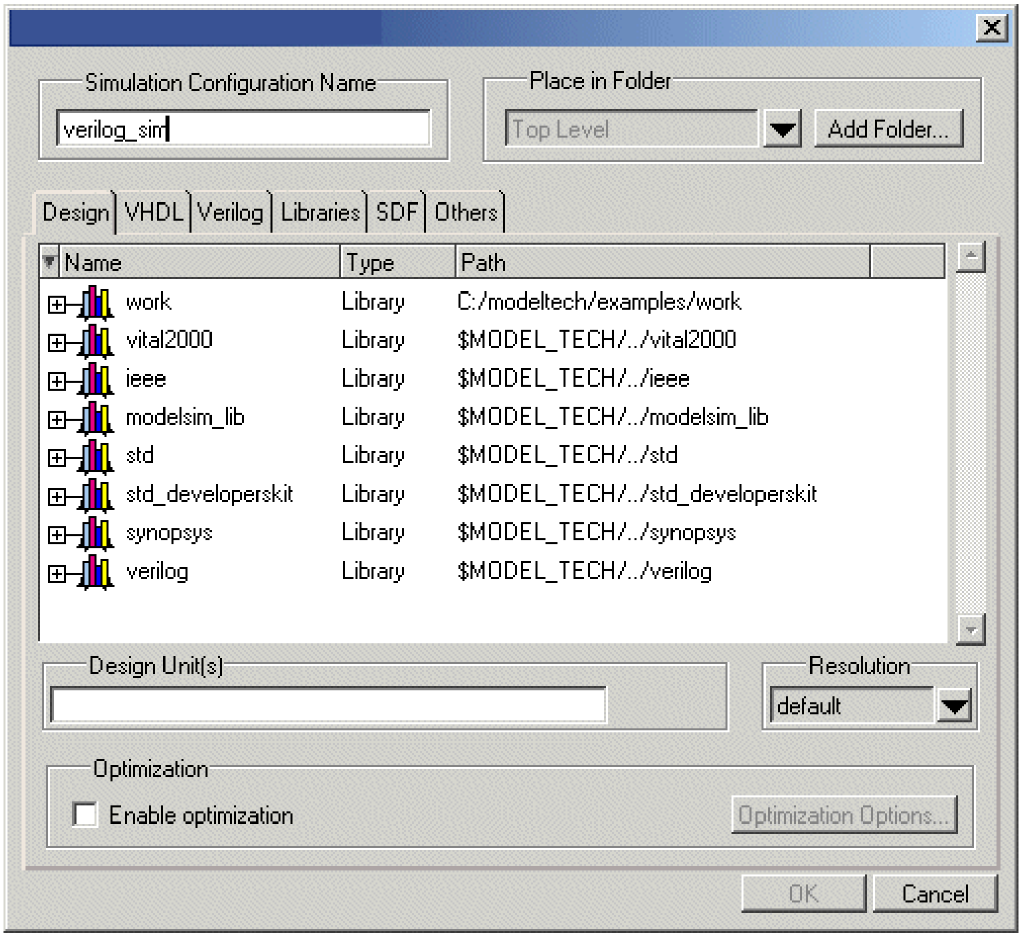
To ungroup files, select the group and click the Ungroup button.

Creating a Simulation Configuration

A Simulation Configuration associates a design unit(s) and its simulation options. For example, say you routinely load a particular design and you have to specify the simulator resolution, generics, and SDF timing files. Ordinarily you would have to specify those options each time you load the design. With a Simulation Configuration, you would specify the design and those options and then save the configuration with a name (e.g., *top_config*). The name is then listed in the Project tab and you can double-click it to load the design along with its options.

To create a Simulation Configuration, follow these steps:

1. Select **File > Add to Project > Simulation Configuration** or select it from the context menu in the Project tab.



Simulation Configuration Name: verilog_sim

Place in Folder: Top Level

Name	Type	Path
work	Library	C:/modeltech/examples/work
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib
std	Library	$MODEL_TECH/../std
std_developerskit	Library	$MODEL_TECH/../std_developerskit
synopsys	Library	$MODEL_TECH/../synopsys
verilog	Library	$MODEL_TECH/../verilog

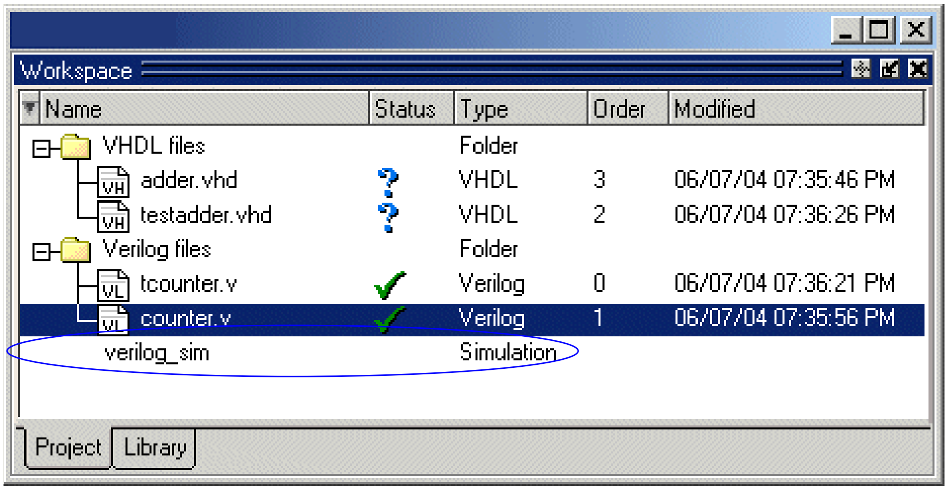
Design Unit(s):

Resolution: default

Optimization: ☐ Enable optimization

2. Specify a name in the **Simulation Configuration Name** field.
3. Specify the folder in which you want to place the configuration (see Organizing Projects with Folders).
4. Select one or more design unit(s). Use the Control and/or Shift keys to select more than one design unit. The design unit names appear in the **Simulate** field when you select them.
5. Use the other tabs in the dialog to specify any required simulation options.

Click OK and the simulation configuration is added to the Project tab.



Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/04 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/04 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/04 07:36:21 PM
counter.v	✓	Verilog	1	06/07/04 07:35:56 PM
verilog_sim		Simulation		

Double-click the Simulation Configuration to load the design.

Optimization Configurations

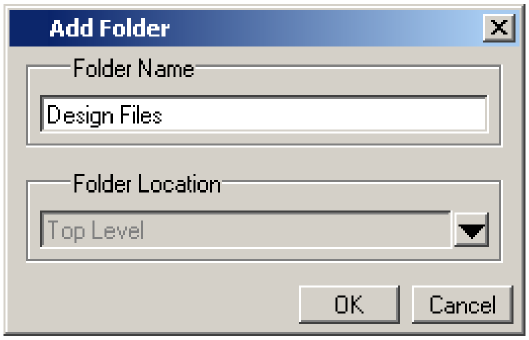
Similar to Simulation Configurations, Optimization Configurations are named objects that represent an optimized simulation. The process for creating and using them is similar to that for Simulation Configurations (see above). You create them by selecting **File** > **Add to Project** > **Optimization Configuration** and specifying various options in a dialog.

Organizing Projects with Folders

The more files you add to a project, the harder it can be to locate the item you need. You can add "folders" to the project to organize your files. These folders are akin to directories in that you can have multiple levels of folders and sub-folders. However, no actual directories are created via the file system–the folders are present only within the project file.

Adding a Folder

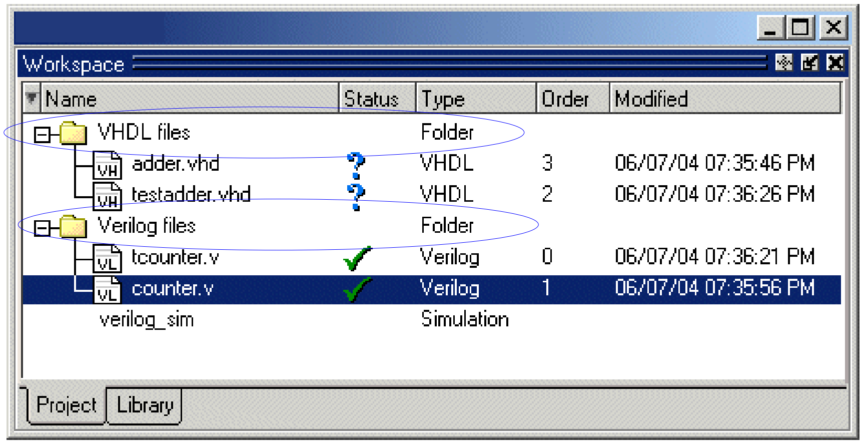
To add a folder to your project, select **File > Add to Project > Folder** or right-click in the Project tab and select **Add to Project > Folder**.



Folder Name: Design Files

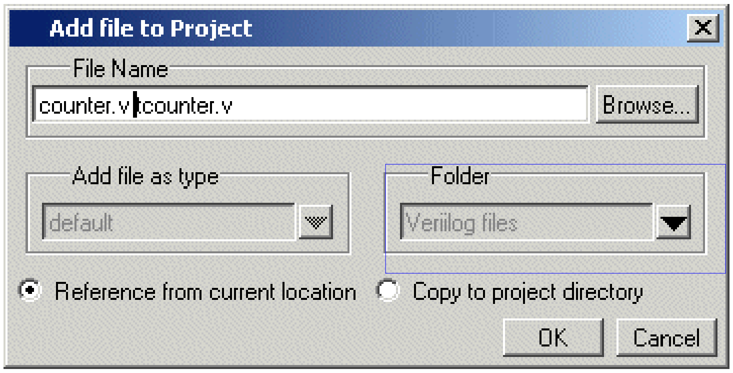
Folder Location: Top Level

Specify the Folder Name, the location for the folder, and click OK. The folder will be displayed in the Project tab.



Name	Status	Type	Order	Modified
VHDL files		Folder		
adder.vhd	?	VHDL	3	06/07/04 07:35:46 PM
testadder.vhd	?	VHDL	2	06/07/04 07:36:26 PM
Verilog files		Folder		
tcounter.v	✓	Verilog	0	06/07/04 07:36:21 PM
counter.v	✓	Verilog	1	06/07/04 07:35:56 PM
verilog_sim		Simulation		

You use the folders when you add new objects to the project. For example, when you add a file, you can select which folder to place it in.



Add file to Project

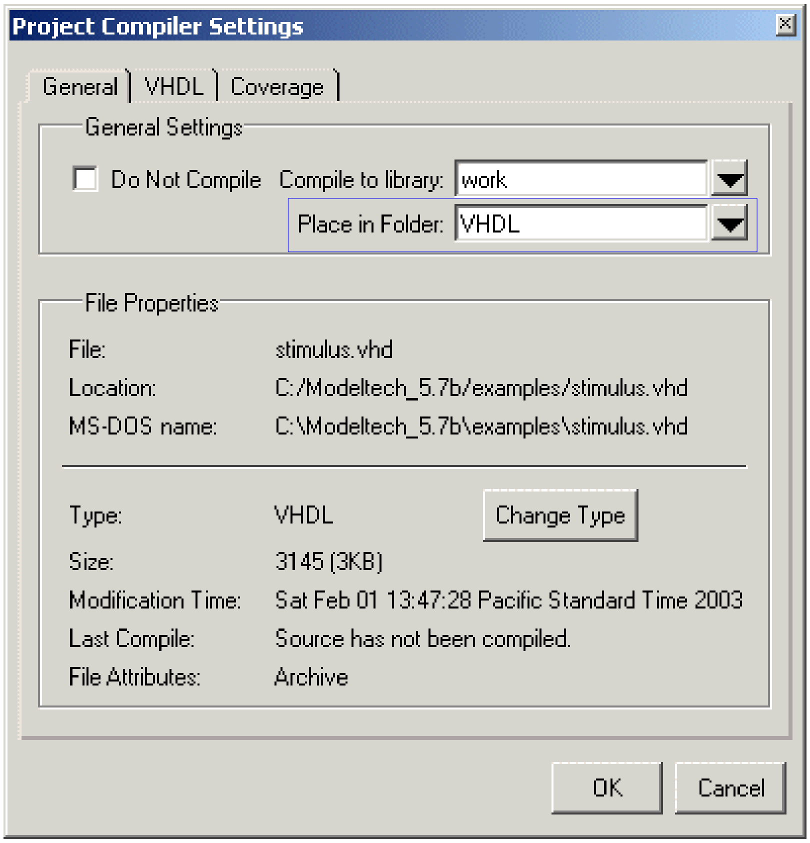
File Name: counter.v counter.v

Add file as type: default

Folder: Verilog files

● Reference from current location ○ Copy to project directory

If you want to move a file into a folder later on, you can do so using the Properties dialog for the file (right-click on the file and select Properties from the context menu).



Project Compiler Settings

General Settings

□ Do Not Compile

Compile to library: work

Place in Folder: VHDL

File Properties

File: stimulus.vhd
Location: C:/Modeltech_5.7b/examples/stimulus.vhd
MS-DOS name: C:\Modeltech_5.7b\examples\stimulus.vhd

Type: VHDL
Size: 3145 (3KB)
Modification Time: Sat Feb 01 13:47:28 Pacific Standard Time 2003
Last Compile: Source has not been compiled.
File Attributes: Archive

On Windows platforms, you can also just drag-and-drop a file into a folder.

Specifying File Properties and Project Settings

You can set two types of properties in a project: file properties and project settings. File properties affect individual files; project settings affect the entire project.

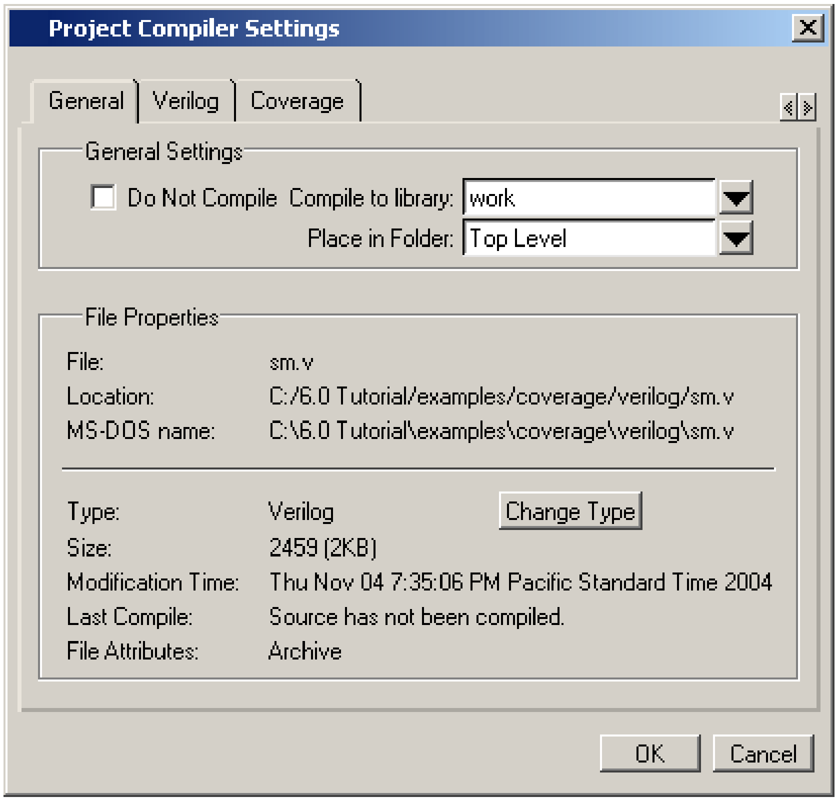
File Compilation Properties

The VHDL and Verilog compilers (**vcom** and **vlog**, respectively) have numerous options that affect how a design is compiled and subsequently simulated. You can customize the settings on individual files or a group of files.

Note

Any changes you make to the compile properties outside of the project, whether from the command line, the GUI, or the *modelsim.ini* file, *will not* affect the properties of files already in the project.

To customize specific files, select the file(s) in the Project tab, right click on the file names, and select **Properties**. The resulting Project Compiler Settings dialog varies depending on the number and type of files you have selected. If you select a single VHDL or Verilog file, you will see the General tab, Coverage tab, and the VHDL or Verilog tab, respectively. If you select a SystemC file, you will see only the General tab. On the General tab, you will see file properties such as Type, Location, and Size. If you select multiple files, the file properties on the General tab are not listed. Finally, if you select both a VHDL file and a Verilog file, you will see all tabs but no file information on the General tab.



Project Compiler Settings

General Settings

□ Do Not Compile Compile to library: work

Place in Folder: Top Level

File Properties

File: sm.v
Location: C:/6.0 Tutorial/examples/coverage/verilog/sm.v
MS-DOS name: C:\6.0 Tutorial\examples\coverage\verilog\sm.v

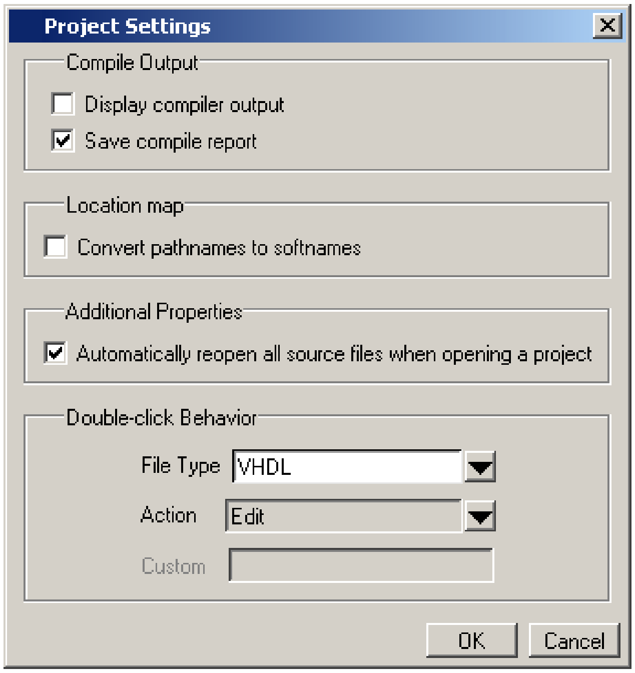
Type: Verilog
Size: 2459 (2KB)
Modification Time: Thu Nov 04 7:35:06 PM Pacific Standard Time 2004
Last Compile: Source has not been compiled.
File Attributes: Archive

When setting options on a group of files, keep in mind the following:

- If two or more files have different settings for the same option, the checkbox in the dialog will be "grayed out." If you change the option, you cannot change it back to a "multi- state setting" without cancelling out of the dialog. Once you click OK, ModelSim will set the option the same for all selected files.
- If you select a combination of VHDL and Verilog files, the options you set on the VHDL and Verilog tabs apply only to those file types.

Project Settings

To modify project settings, right-click anywhere within the Project tab and select **Project Settings**.



Project Settings

Compile Output

□ Display compiler output

☑ Save compile report

Location map

□ Convert pathnames to softnames

Additional Properties

☑ Automatically reopen all source files when opening a project

Double-click Behavior

File Type VHDL

Action Edit

Custom

Accessing Projects from the Command Line

Generally, projects are used from within the ModelSim GUI. However, standalone tools will use the project file if they are invoked in the project's root directory. If you want to invoke outside the project directory, set the **MODELSIM** environment variable with the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

You can also use the project command from the command line to perform common operations on projects.

Chapter 5
Design Libraries

VHDL designs are associated with libraries, which are objects that contain compiled design units. SystemC, Verilog and SystemVerilog designs simulated within ModelSim are compiled into libraries as well.

Design Library Overview

A *design library* is a directory or archive that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, and configurations; and Verilog modules and UDPs (user-defined primitives); and SystemC modules. The design units are classified as follows:

- **Primary design units** — Consist of entities, package declarations, configuration declarations, modules, and UDPs, and SystemC modules. Primary design units within a given library must have unique names.
- **Secondary design units** — Consist of architecture bodies, package bodies, and optimized Verilog modules. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities or modules.

Design Unit Information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

Working Library Versus Resource Libraries

Design libraries can be used in two ways: 1) as a local working library that contains the compiled version of your design; 2) as a resource library. The contents of your working library will change as you update your design and recompile. A resource library is typically static and serves as a parts source for your design. You can create your own resource libraries, or they may be supplied by another design team or a third party (e.g., a silicon vendor).

Only one library can be the working library. In contrast any number of libraries can be resource libraries during a compilation. You specify which resource libraries will be used when the design is compiled, and there are rules to specify in which order they are searched (see Specifying the Resource Libraries).

A common example of using both a working library and a resource library is one where your gate-level design and testbench are compiled into the working library, and the design references gate-level models in a separate resource library.

The Library Named "work"

The library named "work" has special attributes within ModelSim; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units (i.e., it doesn't need to be mapped). In other words the **work** library is the default *working* library.

Archives

By default, design libraries are stored in a directory structure with a sub-directory for each design unit in the library. Alternatively, you can configure a design library to use archives. In this case each design unit is stored in its own archive file. To create an archive, use the **-archive** argument to the vlib command.

Generally you would do this only in the rare case that you hit the reference count limit on I-nodes due to the ".." entries in the lower-level directories (the maximum number of sub-directories on UNIX and Linux is 65533). An example of an error message that is produced when this limit is hit is:

```
mkdir: cannot create directory `65534': Too many links
```

Archives may also have limited value to customers seeking disk space savings.

Note that GMAKE won't work with these archives on the IBM platform.

Working with Design Libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within ModelSim, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names.

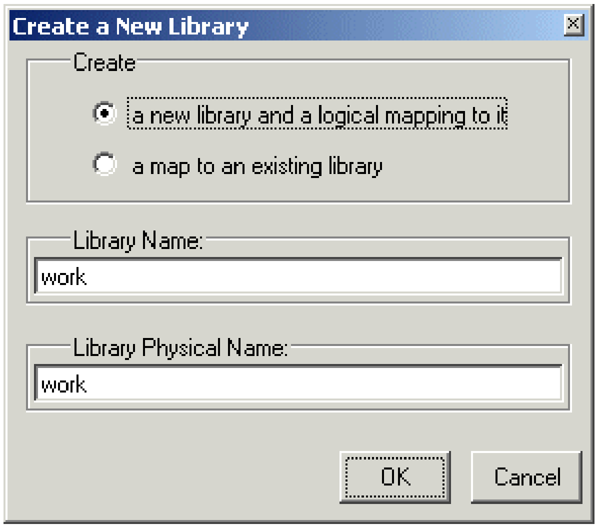
Creating a Library

When you create a project (see Getting Started with Projects), ModelSim automatically creates a working design library. If you don't create a project, you need to create a working design library before you run the compiler. This can be done from either the command line or from the ModelSim graphic interface.

From the ModelSim prompt or a UNIX/DOS prompt, use this vlib command:

```
vlib <directory_pathname>
```

To create a new library with the ModelSim graphic interface, select **File** > **New** > **Library**.



Create a New Library

Create

● a new library and a logical mapping to it

○ a map to an existing library

Library Name: work

Library Physical Name: work

When you click **OK**, ModelSim creates the specified library directory and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a ModelSim library.

The new map entry is written to the *modelsim.ini* file in the [Library] section. See Library Path Variables for more information.

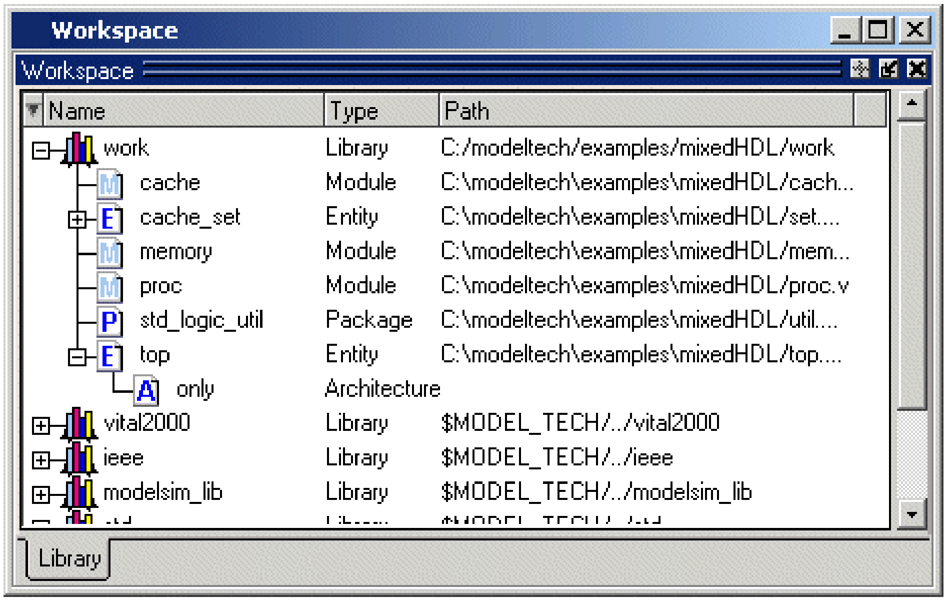
Note

Remember that a design library is a special kind of directory; the only way to create a library is to use the ModelSim GUI or the vlib command. Do not try to create libraries using UNIX, DOS, or Windows commands.

Managing Library Contents

Library contents can be viewed, deleted, recompiled, edited and so on using either the graphic interface or command line.

The Library tab in the Workspace pane provides access to design units (configurations, modules, packages, entities, and architectures, and SystemC modules) in a library. Various information about the design units is displayed in columns to the right of the design unit name.



Name	Type	Path
work	Library	C:/modeltech/examples/mixedHDL/work
cache	Module	C:\modeltech\examples\mixedHDL/cach...
cache_set	Entity	C:\modeltech\examples\mixedHDL/set....
memory	Module	C:\modeltech\examples\mixedHDL/mem...
proc	Module	C:\modeltech\examples\mixedHDL/proc.v
std_logic_util	Package	C:\modeltech\examples\mixedHDL/util....
top	Entity	C:\modeltech\examples\mixedHDL/top....
only	Architecture	
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib

The Library tab has a context menu with various commands that you access by clicking your right mouse button (Windows—2nd button, UNIX—3rd button) in the Library tab.

The context menu includes the following commands:

- **Simulate** — Loads the selected design unit and opens structure and Files tabs in the workspace. Related command line command is vsim.
- **Simulate with Coverage** — Loads the selected design unit and collects code coverage data. Related command line command is vsim **-coverage**.
- **Edit** — Opens the selected design unit in the Source window, or if a library is selected, opens the Edit Library Mapping dialog (see Library Mappings with the GUI).
- **Refresh** — Rebuilds the library image of the selected library without using source code. Related command line command is vcom or vlog with the **-refresh** argument.
- **Recompile** — Recompiles the selected design unit. Related command line command is vcom or vlog.
- **Optimize** — Optimizes a Verilog design unit. Related command line command is vopt.
- **Update** — Updates the display of available libraries and design units.

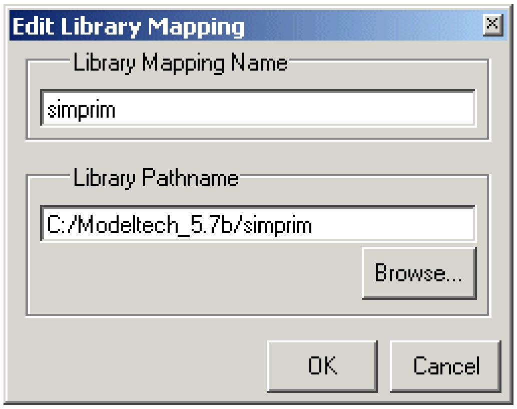
Assigning a Logical Name to a Design Library

VHDL uses logical library names that can be mapped to ModelSim library directories. By default, ModelSim can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the GUI, a command, or a project to assign a logical name to a design library.

Library Mappings with the GUI

To associate a logical name with a library, select the library in the workspace, right-click and select **Edit** from the context menu. This brings up a dialog box that allows you to edit the mapping.



Edit Library Mapping

Library Mapping Name: simprim

Library Pathname: C:/Modeltech_5.7b/simprim

The dialog box includes these options:

- **Library Mapping Name** — The logical name of the library.
- **Library Pathname** — The pathname to the library.

Library Mapping from the Command Line

You can issue a command to set the mapping between a logical library name and a directory; its form is:

```
vmap <logical_name> <directory_pathname>
```

You may invoke this command from either a UNIX/DOS prompt or from the command line within ModelSim.

The vmap command adds the mapping to the library section of the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, use a text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

Unix Symbolic Links

You can also create a UNIX symbolic link to the library using the host platform command:

ln -s <directory_pathname> <logical_name>

The vmap command can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

vmap <logical_name>

Library Search Rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

Moving a Library

Individual design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory or an archive.

Setting Up Libraries for Group Use

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the "others" clause. For example:

```
[library]
```

```
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

Only one "others" clause can be entered in the library section.

Specifying the Resource Libraries

Verilog Resource Libraries

ModelSim supports separate compilation of distinct portions of a Verilog design. The vlog compiler is used to compile one or more source files into a specified library. The library thus contains pre-compiled modules and UDPs that are referenced by the simulator as it loads the design.

Note

Resource libraries are specified differently for Verilog and VHDL. For Verilog you use either the **-L** or **-Lf** argument to vlog. See Library Usage for more information.

VHDL Resource Libraries

Within a VHDL source file, you use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation. The vcom command adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use vcom **-work** and specify the name of the desired target library.

Predefined Libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual, Std 1076*. See also, Using the TextIO Package.

A VHDL **use** clause can be specified to select particular declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every VHDL design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package *standard*, in the design library named *std*, are to be visible to the VHDL design unit immediately following the **use** clause. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

Alternate IEEE Libraries Supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure* — Contains only IEEE approved packages (accelerated for ModelSim).
- *ieee* — Contains precompiled Synopsys and IEEE arithmetic packages which have been accelerated by Model Technology including math_complex, math_real, numeric_bit, numeric_std, std_logic_1164, std_logic_misc, std_logic_textio, std_logic_arith, std_logic_signed, std_logic_unsigned, vital_primitives, and vital_timing.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

Rebuilding Supplied Libraries

Resource libraries are supplied precompiled in the *modeltech* installation directory. If you need to rebuild these libraries, the sources are provided in the *vhdl_src* directory; a macro file is also provided for Windows platforms (*rebldlibs.do*). To rebuild the libraries, invoke the DO file from within ModelSim with this command:

do rbldlibs.do

Make sure your current directory is the *modeltech* install directory before you run this file.

Note

Because accelerated subprograms require attributes that are available only under the 1993 standard, many of the libraries are built using vcom with the **-93** option.

Shell scripts are provided for UNIX (*rebuild_libs.csh* and *rebuild_libs.sh*). To rebuild the libraries, execute one of the *rebuild_libs* scripts while in the *modeltech* directory.

Regenerating Your Design Libraries

Depending on your current ModelSim version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can regenerate your design libraries using the **Refresh** command from the Library tab context menu (see Managing Library Contents), or by using the **-refresh** argument to vcom and vlog.

From the command line, you would use vcom with the **-refresh** option to update VHDL design units in a library, and vlog with the **-refresh** option to update Verilog design units. By default, the work library is updated; use **-work <library>** to update a different library. For example, if you have a library named *mylib* that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh
vlog -work mylib -refresh
```

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of ModelSim. In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches, directives, language constructs, or features that do not exist in the older release.

Note

You don't need to regenerate the *std*, *ieee*, *vital22b*, and *verilog* libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

Maintaining 32- and 64-bit Versions in the Same Library

It is possible with ModelSim to maintain 32-bit and 64-bit versions of a design in the same library, as long as they haven't been optimized by the vopt command.

To do this, you must compile the design with the 32-bit version and then "refresh" the design with the 64-bit version. For example:

Using the 32-bit version of ModelSim:

```
vlog file1.v file2.v -forcecode -work asic_lib
```

Next, using the 64-bit version of ModelSim:

```
vlog -work asic_lib -refresh
```

This allows you to use either version without having to do a refresh.

Do not compile the design with one version, and then recompile it with the other. If you do this, ModelSim will remove the first module, because it could be "stale."

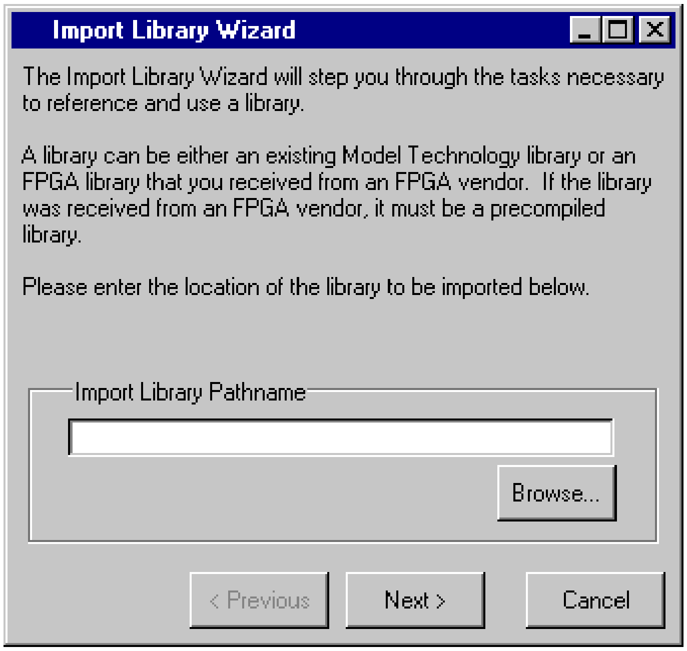
Importing FPGA Libraries

ModelSim includes an import wizard for referencing and using vendor FPGA libraries. The wizard scans for and enforces dependencies in the libraries and determines the correct mappings and target directories.

Note

The FPGA libraries you import must be pre-compiled. Most FPGA vendors supply pre-compiled libraries configured for use with ModelSim.

To import an FPGA library, select **File > Import > Library**.



The Import Library Wizard will step you through the tasks necessary to reference and use a library.

A library can be either an existing Model Technology library or an FPGA library that you received from an FPGA vendor. If the library was received from an FPGA vendor, it must be a precompiled library.

Please enter the location of the library to be imported below.

Import Library Pathname

Follow the instructions in the wizard to complete the import.

Protecting Source Code Using -nodebug

The **-nodebug** argument for both vcom and vlog hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

Note

-nodebug encrypts entire files. The Verilog **`protect** compiler directive allows you to encrypt regions within a file. See Compiler Directives for details.

When you compile with **-nodebug**, all source text, identifiers, and line number information are stripped from the resulting compiled object, so ModelSim cannot locate or display any information of the model except for the external pins. Specifically, this means that:

- a Source window will not display the design units' source code
- a structure pane will not display the internal structure
- the Objects pane will not display internal signals
- the Active Processes pane will not display internal processes
- the Locals pane will not display internal variables
- none of the hidden objects may be accessed through the Dataflow window or with ModelSim commands

You can access the design units comprising your model via the library, and you may invoke vsim directly on any of these design units and see the ports. To restrict even this access in the lower levels of your design, you can use the following **-nodebug** options when you compile:

Table 5-1.

Command and Switch	Result
vcom -nodebug=ports	makes the ports of a VHDL design unit invisible
vlog -nodebug=ports	makes the ports of a Verilog design unit invisible
vlog -nodebug=pli	prevents the use of PLI functions to interrogate the module for information
vlog -nodebug=ports+pli	combines the functions of -nodebug=ports and -nodebug=pli

Don't use the **=ports** option on a design without hierarchy, or on the top level of a hierarchical design. If you do, no ports will be visible for simulation. Rather, compile all lower portions of the design with **-nodebug=ports** first, then compile the top level with **-nodebug** alone.

Design units or modules compiled with **-nodebug** can only instantiate design units or modules that are also compiled **-nodebug**.

Chapter 6
VHDL Simulation

This chapter describes how to compile, optimize, and simulate VHDL designs in ModelSim. It also discusses using the TextIO package with ModelSim*;* ModelSim's implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling; and ModelSim's special built-in utilities package.

The TextIO package is defined within the *VHDL Language Reference Manual, IEEE Std 1076*; it allows human-readable text input from a declared source within a VHDL file during simulation.

Basic VHDL Flow

Simulating VHDL designs with ModelSim includes four general steps:

1. Compile your VHDL code into one or more libraries using the vcom command. See Compiling VHDL Files for details.
2. Elaborate and optimize your design using the vopt command. See Chapter 3, Optimizing Designs with vopt for details.
3. Load your design with the vsim command. See Simulating VHDL Designs for details.
4. Run and debug your design.

Compiling VHDL Files

Creating a Design Library for VHDL

Before you can compile your source files, you must create a library in which to store the compilation results. Use vlib to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX, MS Windows, or DOS commands – always use the vlib command.

See Design Libraries for additional information on working with libraries.

Invoking the VHDL Compiler

ModelSim compiles one or more VHDL design units with a single invocation of vcom, the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with 1076 -1987, 1076 -1993, and 1076-2002 versions of VHDL. To do so you will need to compile units from each VHDL version separately. The vcom command compiles using 1076 -2002 rules by default; use the **-87** or **-93** argument to vcom to compile units written with version 1076-1987 or 1076 -1993, respectively. You can also change the default by modifying the VHDL93 variable in the *modelsim.ini* file (see Control Variables Located in INI Files for more information).

Dependency Checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. vcom determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

Range and Index Checking

A range check verifies that a scalar value defined with a range subtype is always assigned a value within its range. An index check verifies that whenever an array subscript expression is evaluated, the subscript will be within the array's range.

Range and index checks are performed by default when you compile your design. You can disable range checks (potentially offering a performance advantage) and index checks using arguments to the vcom command. Or, you can use the **NoRangeCheck** and **NoIndexCheck** variables in the *modelsim.ini* file to specify whether or not they are performed. See Control Variables Located in INI Files.

Range checks in ModelSim are slightly more restrictive than those specified by the VHDL LRM. ModelSim requires any assignment to a signal to also be in range whereas the LRM requires only that range checks be done whenever a signal is updated. Most assignments to signals update the signal anyway, and the more restrictive requirement allows ModelSim to generate better error messages.

Subprogram Inlining

ModelSim attempts to inline subprograms at compile time to improve simulation performance. This happens automatically and should be largely transparent. However, you can disable automatic inlining two ways:

- Invoke vcom with the -O0 or -O1 argument
- Use the *mti_inhibit_inline* attribute as described below

Single-stepping through a simulation varies slightly depending on whether inlining occurred. When single-stepping to a subprogram call that has not been inlined, the simulator stops first at the line of the call, and then proceeds to the line of the first executable statement in the called subprogram. If the called subprogram has been inlined, the simulator does not first stop at the subprogram call, but stops immediately at the line of the first executable statement.

mti_inhibit_inline Attribute

You can disable inlining for individual design units (a package, architecture, or entity) or subprograms with the *mti_inhibit_inline* attribute. Follow these rules to use the attribute:

- Declare the attribute within the design unit's scope as follows:

  ```
  attribute mti_inhibit_inline : boolean;
  ```

- Assign the value true to the attribute for the appropriate scope. For example, to inhibit inlining for a particular function (e.g., "foo"), add the following attribute assignment:

  ```
  attribute mti_inhibit_inline of foo : procedure is true;
  ```

  To inhibit inlining for a particular package (e.g., "pack"), add the following attribute assignment:

  ```
  attribute mti_inhibit_inline of pack : package is true;
  ```

  Do similarly for entities and architectures.

Differences Between Language Versions

There are three versions of the IEEE VHDL 1076 standard: VHDL-1987, VHDL-1993, and VHDL-2002. The default language version for ModelSim is VHDL-2002. If your code was written according to the ’87 or ’93 version, you may need to update your code or instruct ModelSim to use the earlier versions’ rules.

To select a specific language version, do one of the following:

- Select the appropriate version from the compiler options menu in the GUI
- Invoke vcom using the argument -87, -93, or -2002
- Set the VHDL93 variable in the [vcom] section of the *modelsim.ini* file. Appropriate values for VHDL93 are:

  - 0, 87, or 1987 for VHDL-1987

  - 1, 93, or 1993 for VHDL-1993

- 2, 02, or 2002 for VHDL-2002

The following is a list of language incompatibilities that may cause problems when compiling a design.

- VHDL-93 and VHDL-2002 — The only major problem between VHDL-93 and VHDL-2002 is the addition of the keyword "PROTECTED". VHDL-93 programs which use this as an identifier should choose a different name.

  All other incompatibilities are between VHDL-87 and VHDL-93.

- VITAL and SDF — It is important to use the correct language version for VITAL. VITAL2000 must be compiled with VHDL-93 or VHDL-2002. VITAL95 must be compiled with VHDL-87. A typical error message that indicates the need to compile under language version VHDL-87 is:

  ```
  "VITALPathDelay DefaultDelay parameter must be locally static"
  ```

- Purity of NOW — In VHDL-93 the function "now" is impure. Consequently, any function that invokes "now" must also be declared to be impure. Such calls to "now" occur in VITAL. A typical error message:

  ```
  "Cannot call impure function 'now' from inside pure function
  '<name>'"
  ```

- Files — File syntax and usage changed between VHDL-87 and VHDL-93. In many cases vcom issues a warning and continues:

  ```
  "Using 1076-1987 syntax for file declaration."
  ```

  In addition, when files are passed as parameters, the following warning message is produced:

  ```
  "Subprogram parameter name is declared using VHDL 1987 syntax."
  ```

  This message often involves calls to endfile(<name>) where <name> is a file parameter.

- Files and packages — Each package header and body should be compiled with the same language version. Common problems in this area involve files as parameters and the size of type CHARACTER. For example, consider a package header and body with a procedure that has a file parameter:

  ```
  procedure proc1 ( out_file : out std.textio.text) ...
  ```

  If you compile the package header with VHDL-87 and the body with VHDL-93 or VHDL-2002, you will get an error message such as:

  ```
  "** Error: mixed_package_b.vhd(4): Parameter kinds do not conform
  between declarations in package header and body: 'out_file'."
  ```

- Direction of concatenation — To solve some technical problems, the rules for direction and bounds of concatenation were changed from VHDL-87 to VHDL-93. You won't see any difference in simple variable/signal assignments such as:

```
v1 := a & b;
```

But if you (1) have a function that takes an unconstrained array as a parameter, (2) pass a concatenation expression as a formal argument to this parameter, and (3) the body of the function makes assumptions about the direction or bounds of the parameter, then you will get unexpected results. This may be a problem in environments that assume all arrays have "downto" direction.

- xnor — "xnor" is a reserved word in VHDL-93. If you declare an xnor function in VHDL-87 (without quotes) and compile it under VHDL-2002, you will get an error message like the following:

  ```
  ** Error: xnor.vhd(3): near "xnor": expecting: STRING IDENTIFIER
  ```

- 'FOREIGN attribute — In VHDL-93 package STANDARD declares an attribute 'FOREIGN. If you declare your own attribute with that name in another package, then ModelSim issues a warning such as the following:

  ```
  -- Compiling package foopack

  ** Warning: foreign.vhd(9): (vcom-1140) VHDL-1993 added a definition
  of the attribute foreign to package std.standard. The attribute is
  also defined in package 'standard'. Using the definition from
  package 'standard'.
  ```

- Size of CHARACTER type — In VHDL-87 type CHARACTER has 128 values; in VHDL-93 it has 256 values. Code which depends on this size will behave incorrectly. This situation occurs most commonly in test suites that check VHDL functionality. It's unlikely to occur in practical designs. A typical instance is the replacement of warning message:

  ```
  "range nul downto del is null"
  ```

  by

  ```
  "range nul downto 'ÿ' is null" -- range is nul downto y(umlaut)
  ```

- bit string literals — In VHDL-87 bit string literals are of type bit_vector. In VHDL-93 they can also be of type STRING or STD_LOGIC_VECTOR. This implies that some expressions that are unambiguous in VHDL-87 now become ambiguous is VHDL-93. A typical error message is:

  ```
  ** Error: bit_string_literal.vhd(5): Subprogram '=' is ambiguous.
  Suitable definitions exist in packages 'std_logic_1164' and
  'standard'.
  ```

- Sub-element association — In VHDL-87 when using individual sub-element association in an association list, associating individual sub-elements with NULL is discouraged. In VHDL-93 such association is forbidden. A typical message is:

  ```
  "Formal '<name>' must not be associated with OPEN when subelements
  are associated individually."
  ```

Simulating VHDL Designs

A VHDL design is ready for simulation after it has been compiled with **vcom** and possibly optimized with **vopt** (see Optimizing Designs with vopt). The simulator may then be invoked with the name of the configuration or entity/architecture pair or the name you assigned to the optimized version of the design.

Note

This section discusses simulation from the UNIX or Windows/DOS command line. You can also use a project to simulate (see Getting Started with Projects) or the **Simulate** dialog box.

This example invokes vsim on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

vsim is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking vsim with the **-sdfmin**, **-sdftyp**, or **-sdfmax** option. Using the SDF file *f1.sdf* in the current work directory, the following invocation of vsim annotates maximum timing values for the design unit *my_asic*:

```
vsim -sdfmax /my_asic=f1.sdf my_asic
```

By default, the timing checks within VITAL models are enabled. They can be disabled with the **+notimingchecks** option. For example:

```
vsim +notimingchecks topmod
```

Simulator Resolution Limit (VHDL)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The default resolution limit is set to the value specified by the Resolution variable in the *modelsim.ini* file. You can view the current resolution by invoking the report command with the **simulator state** option.

Overriding the Resolution

You can override ModelSim's default resolution by specifying the **-t** option on the command line or by selecting a different Simulator Resolution in the **Simulate** dialog box. Available resolutions are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

For example this command chooses 10 ps resolution:

```
vsim -t 10ps topmod
```

Clearly you need to be careful when doing this type of operation. If the resolution set by **-t** is larger than a delay value in your design, the delay values in that design unit are rounded to the closest multiple of the resolution. In the example above, a delay of 4 ps would be rounded to 0 ps.

Choosing the Resolution for VHDL

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Default Binding

By default ModelSim performs default binding when you load the design with vsim. The advantage of performing default binding at load time is that it provides more flexibility for compile order. Namely, entities don't necessarily have to be compiled before other entities/architectures which instantiate them.

However, you can force ModelSim to perform default binding at compile time. This may allow you to catch design errors (e.g., entities with incorrect port lists) earlier in the flow. Use one of these two methods to change when default binding occurs:

- Specify the **-bindAtCompile** argument to vcom
- Set the BindAtCompile variable in the *modelsim.ini* to 1 (true)

Default Binding Rules

When looking for an entity to bind with, ModelSim searches the currently visible libraries for an entity with the same name as the component. ModelSim does this because IEEE 1076-1987 contained a flaw that made it almost impossible for an entity to be directly visible if it had the same name as the component. In short, if a component was declared in an architecture, any like-named entity above that declaration would be hidden because component/entity names cannot be overloaded. As a result we implemented the following rules for determining default binding:

- If performing default binding at load time, search the libraries specified with the **-Lf** argument to **vsim**.
- If a directly visible entity has the same name as the component, use it.
- If an entity would be directly visible in the absence of the component declaration, use it.
- If the component is declared in a package, search the library that contained the package for an entity with the same name.

If none of these methods is successful, ModelSim will also do the following:

- Search the work library.
- Search all other libraries that are currently visible by means of the **library** clause.

- If performing default binding at load time, search the libraries specified with the **-L** argument to **vsim**.

Note that these last three searches are an extension to the 1076 standard.

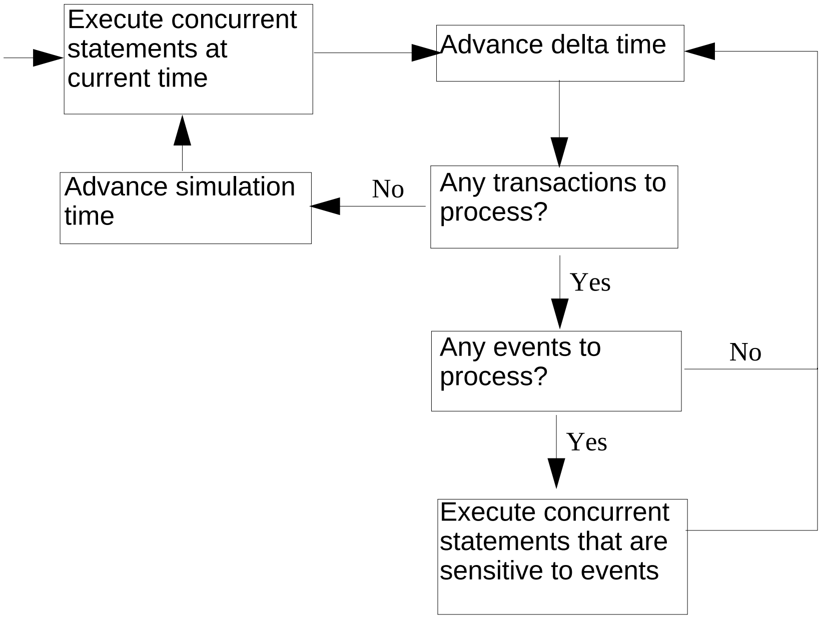
Disabling Default Binding

If you want default binding to occur only via configurations, you can disable ModelSim's normal default binding methods by setting the RequireConfigForAllDefaultBinding variable in the *modelsim.ini* to 1 (true).

Delta Delays

Event-based simulators such as ModelSim may process many events at a given simulation time. Multiple signals may need updating, statements that are sensitive to these signals must be executed, and any new events that result from these statements must then be queued and executed as well. The steps taken to evaluate the design without advancing simulation time are referred to as "delta times" or just "deltas."

The diagram below represents the process for VHDL designs. This process continues until the end of simulation time.


Execute concurrent statements at current time
Advance delta time
Advance simulation time
No
Any transactions to process?
Yes
Any events to process?
No
Yes
Execute concurrent statements that are sensitive to events

This mechanism in event-based simulators may cause unexpected results. Consider the following code snippet:

```
    clk2 <= clk;

    process (rst, clk)
    begin
      if(rst = '0')then
        s0 <= '0';
      elsif(clk'event and clk='1') then
        s0 <= inp;
      end if;
    end process;

  process (rst, clk2)
    begin
      if(rst = '0')then
        s1 <= '0';
      elsif(clk2'event and clk2='1') then
        s1 <= s0;
      end if;
    end process;
```

In this example you have two synchronous processes, one triggered with *clk* and the other with *clk2*. To your surprise, the signals change in the *clk2* process on the same edge as they are set in the *clk* process. As a result, the value of *inp* appears at *s1* rather than *s0*.

During simulation an event on *clk* occurs (from the testbench). From this event ModelSim performs the "clk2 <= clk" assignment and the process which is sensitive to *clk*. Before advancing the simulation time, ModelSim finds that the process sensitive to *clk2* can also be run. Since there are no delays present, the effect is that the value of *inp* appears at *s1* in the same simulation cycle.

In order to get the expected results, you must do one of the following:

- Insert a delay at every output
- Make certain to use the same clock
- Insert a delta delay

To insert a delta delay, you would modify the code like this:

```
    process (rst, clk)
      begin
        if(rst = '0')then
          s0 <= '0';
        elsif(clk'event and clk='1') then
          s0 <= inp;
          s0_delayed <= s0;
        end if;
      end process;

     process (rst, clk2)
      begin
        if(rst = '0')then
          s1 <= '0';
        elsif(clk2'event and clk2='1') then
          s1 <= s0_delayed;
        end if;
      end process;
```

The best way to debug delta delay problems is observe your signals in the List window. There you can see how values change at each delta time.

Detecting Infinite Zero-Delay Loops

If a large number of deltas occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, ModelSim defines a limit, the “iteration limit", on the number of successive deltas that can occur. When ModelSim reaches the iteration limit, it issues a warning message.

The iteration limit default value is 5000 . If you receive an iteration limit warning, first increase the iteration limit and try to continue simulation. You can set the iteration limit from the **Simulate** > **Runtime Options** menu or by modifying the IterationLimit variable in the *modelsim.ini*. See Control Variables Located in INI Files for more information on modifying the *modelsim.ini* file.

If the problem persists, look for zero-delay loops. Run the simulation and look at the source code when the error occurs. Use the step button to step through the code and see which signals or variables are continuously oscillating. Two common causes are a loop that has no exit, or a series of gates with zero delay where the outputs are connected back to the inputs.

Simulating VHDL with an Elaboration File

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

You can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an Elaboration File?

In many cases design loading time is not that important. For example, if you’re doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you’re comparing like to like.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration File Flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

1. If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you’ll have to use $sdf_annotate system tasks. Note that use of $sdf_annotate causes timing to be applied after elaboration.
2. Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see Modifying Stimulus below).
3. Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an Elaboration File

Elaboration file creation is performed with the same **vsim** settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time, as detailed below.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to vsim.

The **-elab_cont** argument is used to create the elaboration file then continue with the simulation after the elaboration file is created. You can use the **-c** switch with **-elab_cont** to continue the simulation in command-line mode.

Note

Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an Elaboration File

To load an elaboration file, use the **-load_elab <filename>** argument to vsim. By default the elaboration file will load in command-line mode or interactive mode depending on the argument (-c or -i) used during elaboration file creation. If no argument was used during creation, the -load_elab argument will default to the interactive mode.

The **vsim** arguments listed below can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

All other vsim arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

Note

The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, the same version of any PLI/FLI code loaded in the simulation, and and the same release of ModelSim.

Modifying Stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can't be used to redirect stdin/stdout.
- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using With the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and the PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the Foreign Language Interface Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the vsim command for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design, use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling, and reloading the entire design, ModelSim allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling, and reloading the entire design is as follows:

vsim -load_elab <filename> -filemap_elab vectors=alt_vectors

Checkpointing and Restoring VHDL Simulations

The checkpoint and restore commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Table 6-1.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current **vsim** session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous invocation of **vsim**	vsim -restore <filename>

Checkpoint File Contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file

- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **$fopen** system task
- state of foreign architectures
- state of PLI/VPI/DPI code

Checkpoint Exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the Foreign Language Interface Reference Manual or Verilog PLI / VPI / DPI for more information.

Controlling Checkpoint File Compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

set CheckpointCompressMode 0

To turn compression back on, use this command:

set CheckpointCompressMode 1

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The Difference Between Checkpoint/restore and Restart

The restart command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog $fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart,** however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using Macros with Restart and Checkpoint/restore

The restart command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a checkpoint and later in the same session doing a restore of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Checkpointing Foreign C Code That Works with Heap Memory

If checkpointing foreign C code (FLI/PLI/VPI/DPI) that works with heap memory, use mti_Malloc() rather than raw malloc() or new. Any memory allocated with mti_Malloc() is guaranteed to be restored correctly. Any memory allocated with raw malloc() will not be restored correctly, and simulator crashes can result.

Checkpointing a Running Simulation

In general you can invoke a checkpoint command only when the simulation is stopped. If you need to checkpoint without stopping the simulation, you need to write a script that utilizes the when command and variables from your code to trigger a checkpoint. The example below show how this might be done with a simple Verilog design.

Keep in mind that the variable(s) in your code must be visible at the simulation time that the checkpoint will occur. Some global optimizations performedby ModelSim may limit variable visibility, and you may need to optimize your design using the **+acc** argument to vopt.

You would compile and run the example like this:

```
vlog when.v
vsim -c when -do "do when.do"
```

where *when.do* is:

```
onbreak {
    echo "Resume macro at $now"
    resume
}
quietly set continueSim 1
quietly set whenFired 0
quietly set checkpointCntr 0

when { needToSave = 1 } {
    echo "when Stopping to allow checkpoint at $now"
    set whenFired 1
    stop
}

while {$continueSim} {
    run -all
    if { $whenFired} {
       set whenFired 0
       echo "Out of run command. Do checkpoint here"
       checkpoint cpf.n[incr checkpointCntr].cpt
    }
}
```

and *when.v* is:

```
module when;

   reg clk;
   reg [3:0] cnt;
   reg  needToSave;

   initial
     begin
        needToSave = 0;
        clk = 0;
       cnt = 0;
       #1000;
       $display("Done at time %t", $time);
       $finish;
    end

   always #10 clk = ~clk;

   always @(posedge clk)
     begin
     cnt = cnt + 1;
     if (cnt == 4'hF)
       begin
          $display( "Need to Save : %b", needToSave);
          needToSave = 1;
       end
     end

   // Need to reset the flag, but must wait a timestep
   // so that the when command has a chance to fire

   always @(posedge needToSave)
     #1 needToSave = 0;
endmodule
```

and the transcript output is:

```
# vsim -do {do when.do} -c when
# //  ModelSim SE 6.1 Beta Feb 26 2005 Linux 2.6.9-5.0.3.ELhugemem
# //
# //  Copyright Mentor Graphics Corporation 2005
# //              All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.when
# do when.do
# Need to Save : 0
# when Stopping to allow checkpoint at 290 # Simulation stop requested.
# Resume macro at 290
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 610
# Simulation stop requested.
# Resume macro at 610
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 930
# Simulation stop requested.
# Resume macro at 930
# Out of run command. Do checkpoint here
# Done at time                 1000
# ** Note: $finish    : when.v(14)
#    Time: 1 us  Iteration: 0  Instance: /when
```

Using the TextIO Package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
   PROCESS
      VARIABLE i: INTEGER:= 42;
      VARIABLE LLL: LINE;
   BEGIN
      WRITE (LLL, i);
      WRITELINE (OUTPUT, LLL);
      WAIT;
   END PROCESS;
END simple_behavior;
```

Syntax for File Declaration

The VHDL’87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ]  file_logical_name ;
```

where "file_logical_name" must be a string expression.

In newer versions of the 1076 spec, syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

where "file_open_information" is:

```
[open file_open_kind_expression] is file_logical_name
```

You can specify a full or relative path as the file_logical_name; for example (VHDL’87):

```
file filename : TEXT is in "/usr/rick/myfile";
```

Normally if a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNs from the subprogram. Alternatively, the opening of files can be delayed until the first read or write by setting the **DelayFileOpen** variable in the *modelsim.ini* file. Also, the number of concurrently open files can be controlled by the **ConcurrentFileLimit** variable. These variables help you manage a large number of files during simulation. See Simulator Variables for more details.

Using STD_INPUT and STD_OUTPUT Within the Tool

The standard VHDL'87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";
file output: TEXT is out "STD_OUTPUT";
```

Updated versions of the TextIO package contain these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD_INPUT is a file_logical_name that refers to characters that are entered interactively from the keyboard, and STD_OUTPUT refers to text that is displayed on the screen.

In ModelSim, reading from the STD_INPUT file allows you to enter text into the current buffer from a prompt in the Transcript pane. The lines written to the STD_OUTPUT file appear in the Transcript.

TextIO Implementation Issues

Writing Strings and Aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT_VECTOR. These lines are reproduced here:

```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;
   JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE(L: inout LINE; VALUE: in STRING;
   JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string’("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE_STRING procedure in the io_utils package, which is located in the file *<install_dir>/modeltech/examples/misc/io_utils.vhd*.

Reading and Writing Hexadecimal Numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, ModelSim supplies hexadecimal routines in the package io_utils, which is located in the file *<install_dir>/modeltech/examples/misc/io_utils.vhd*. To use these routines, compile the io_utils package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;
use work.io_utils.all;
```

Dangling Pointers

Dangling pointers are easily created when using the TextIO package, because WRITELINE de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

Bad VHDL (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := L1;                 -- Copy pointers
WRITELINE (outfile, L1);  -- Deallocate buffer
```

Good VHDL (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := new string’(L1.all);  -- Copy contents
WRITELINE (outfile, L1);    -- Deallocate buffer
```

The ENDLINE Function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access values must be passed as variables, but functions do not allow variable parameters.

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L’LENGTH = 0)
```

The ENDFILE Function

In the *VHDL Language Reference Manuals,* the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

Using Alternative Input/Output Files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT. For example, for an input file:

The VHDL’87 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

The VHDL'93 declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

Flushing the TEXTIO Buffer

Flushing of the TEXTIO buffer is controlled by the UnbufferedOutput variable in the *modelsim.ini* file.

Providing Stimulus

You can stimulate and test a design by reading vectors from a file, using them to drive values onto signals, and testing the results. A VHDL test bench has been included with the ModelSim install files as an example. Check for this file:

```
<install_dir>/modeltech/examples/misc/stimulus.vhd
```

VITAL Specification and Source Code

VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service
445 Hoes Lane
Piscataway, NJ 08854-1331

Tel: (732) 981-0060
Fax: (732) 981-1721
home page: http://www.ieee.org

VITAL source code

The source code for VITAL packages is provided in the directories:

```
/<install_dir>/vhdl_src/vital2.2b
                        /vital95
                        /vital2000
```

VITAL Packages

VITAL 1995 accelerated packages are pre-compiled into the **ieee** library in the installation directory. VITAL 2000 accelerated packages are pre-compiled into the **vital2000** library. If you need to use the newer library, you either need to change the ieee library mapping or add a **use** clause to your VHDL code to access the VITAL 2000 packages.

To change the ieee library mapping, issue the following command:

```
vmap ieee <modeltech>/vital2000
```

Or, alternatively, add use clauses to your code:

```
LIBRARY vital2000;
USE vital2000.vital_primitives.all;
USE vital2000.vital_timing.all;
USE vital2000.vital_memory.all;
```

Note that if your design uses two libraries -one that depends on vital95 and one that depends on vital2000 - then you will have to change the references in the source code to vital2000. Changing the library mapping will not work.

VITAL Compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. ModelSim is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, ModelSim accelerates the VITAL_Timing, VITAL_Primitives, and VITAL_memory packages. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL 1995 and 2000).

VITAL Compliance Checking

Compliance checking is important in enabling VITAL acceleration; to qualify for global acceleration, an architecture must be VITAL-level-one compliant. vcom automatically checks for VITAL 2000 compliance on all entities with the VITAL_Level0 attribute set, and all architectures with the VITAL_Level0 or VITAL_Level1 attribute set.

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking vcom with the option **-novitalcheck**.

You can turn off compliance checking for VITAL 1995 and VITAL 2000 as well, but we strongly suggest that you leave checking on to ensure optimal simulation.

VITAL Compliance Warnings

The following LRM errors are printed as warnings (if they were considered errors they would prevent VITAL level 1 acceleration); they do not affect how the architecture behaves.

- Starting index constraint to DataIn and PreviousDataIn parameters to VITALStateTable do not match (1076.4 section 6.4.3.2.2)
- Size of PreviousDataIn parameter is larger than the size of the DataIn parameter to VITALStateTable (1076.4 section 6.4.3.2.2)
- Signal q_w is read by the VITAL process but is NOT in the sensitivity list (1076.4 section 6.4.3)

The first two warnings are minor cases where the body of the VITAL 1995 LRM is slightly stricter than the package portion of the LRM. Since either interpretation will provide the same simulation results, we chose to make these two cases warnings.

The last warning is a relaxation of the restriction on reading an internal signal that is not in the sensitivity list. This is relaxed only for the CheckEnabled parameters of the timing checks, and only if they are not read elsewhere.

You can control the visibility of VITAL compliance-check warnings in your vcom transcript. They can be suppressed by using the **vcom -nowarn** switch as in **vcom -nowarn 6**. The 6 comes from the warning level printed as part of the warning, i.e., ** WARNING: [6]. You can also add the following line to your *modelsim.ini* file in the VHDL Compiler Control Variables section.

```
[vcom]
Show_VitalChecksWarnings = 0
```

Compiling and Simulating with Accelerated VITAL Packages

vcom automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Optimization occurs on two levels:

- **VITAL Level-0 optimization** — This is a function-by-function optimization. It applies to all level-0 architectures, and any level-1 architectures that failed level-1 optimization.
- **VITAL Level-1 optimization** — Performs global optimization on a VITAL 3.0 level-1 architecture that passes the VITAL compliance checker. This is the default behavior. Note that your models will run faster but at the cost of not being able to see the internal workings of the models.

Compiler Options for VITAL Optimization

Several vcom options control and provide feedback on VITAL optimization:

- -novital

  Causes **vcom** to use VHDL code for VITAL procedures rather than the accelerated and optimized timing and primitive packages. Allows breakpoints to be set in the VITAL behavior process and permits single stepping through the VITAL procedures to debug your model. Also, all of the VITAL data can be viewed in the Locals or Objects pane.

- -O0 | -O4

  Lowers the optimization to a minimum with **-O0** (capital oh zero). Optional. Use this to work around bugs, increase your debugging visibility on a specific cell, or when you want to place breakpoints on source lines that have been optimized out.

  Enable optimizations with **-O4** (default).

- -debugVA

  Prints a confirmation if a VITAL cell was optimized, or an explanation of why it was not, during VITAL level-1 acceleration.

ModelSim VITAL built-ins will be updated in step with new releases of the VITAL packages.

Util Package

The util package serves as a container for various VHDL utilities. The package is part of the modelsim_lib library which is located in the modeltech tree and is mapped in the default *modelsim.ini* file.

To access the utilities in the package, you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

get_resolution

get_resolution returns the current simulator resolution as a real number. For example, 1 femtosecond corresponds to 1e-15.

Syntax

resval := get_resolution;

Returns

Name	Type	Description
resval	real	The simulator resolution represented as a real

Arguments

None

Related functions

- to_real()
- to_time()

Example

If the simulator resolution is set to 10ps, and you invoke the command:

```
resval := get_resolution;
```

the value returned to resval would be 1e-11.

init_signal_driver()

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_driver for complete details.

init_signal_spy()

The init_signal_spy() utility mirrors the value of a VHDL signal or Verilog register/net onto an existing VHDL signal or Verilog register. This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

See init_signal_spy for complete details.

signal_force()

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_force works the same as the force command with the exception that you cannot issue a repeating force.

See signal_force for complete details.

signal_release()

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register or net. This allows you to release signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench). A signal_release works the same as the noforce command.

See signal_release for complete details.

to_real()

to_real() converts the physical type time value into a real value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 1900 fs to a real and the simulator resolution was ps, then the real value would be 2.0 (i.e., 2 ps).

Syntax

```
realval := to_real(timeval);
```

Returns

Name	**Type**	**Description**
realval	real	The time value represented as a real with respect to the simulator resolution

Arguments

Name	**Type**	**Description**
timeval	time	The value of the physical type time

Related functions

- get_resolution
- to_time()

Example

If the simulator resolution is set to ps, and you enter the following function:

```
realval := to_real(12.99 ns);
```

then the value returned to realval would be 12990.0. If you wanted the returned value to be in units of nanoseconds (ns) instead, you would use the get_resolution function to recalculate the value:

```
realval := 1e+9 * (to_real(12.99 ns)) * get_resolution();
```

If you wanted the returned value to be in units of femtoseconds (fs), you would enter the function this way:

```
realval := 1e+15 * (to_real(12.99 ns)) * get_resolution();
```

to_time()

to_time() converts a real value into a time value with respect to the current simulator resolution. The precision of the converted value is determined by the simulator resolution. For example, if you were converting 5.9 to a time and the simulator resolution was ps, then the time value would be 6 ps.

Syntax

```
timeval := to_time(realval);
```

Returns

Name	Type	Description
timeval	time	The real value represented as a physical type time with respect to the simulator resolution

Arguments

Name	Type	Description
realval	real	The value of the type real

Related functions

- get_resolution
- to_real()

Example

If the simulator resolution is set to ps, and you enter the following function:

```
timeval := to_time(72.49);
```

then the value returned to timeval would be 72 ps.

Foreign Language Interface

Foreign language interface (FLI) routines are C programming language functions that provide procedural access to information within Model Technology's HDL simulator, vsim. A user-written application can use these functions to traverse the hierarchy of an HDL design, get information about and set the values of VHDL objects in the design, get information about a simulation, and control (to some extent) a simulation run.

ModelSim’s FLI interface is described in detail in the Foreign Language Interface Reference Manual.

Modeling Memory

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate enough storage.
- Or, you may get very long load, elaboration, or run times.

These problems are usually explained by the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which needs to be loaded or initialized before your simulation starts.

Modeling memory with variables or protected types instead provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

In the VHDL example below, we illustrate three alternative architectures for entity *memory*:

- Architecture *bad_style_87* uses a vhdl signal to store the ram data.
- Architecture *style_87* uses variables in the *memory* process
- Architecture *style_93* uses variables in the architecture.

For large memories, architecture *bad_style_87* runs many times longer than the other two, and uses much more memory. This style should be avoided.

Architectures *style_87* and *style_93* work with equal efficiently. However, VHDL 1993 offers additional flexibility because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

For completeness sake we also show an example using VHDL 2002 protected types, though in this example, protected types offer no advantage over shared variables.

VHDL87 and VHDL93 Example

```
library ieee;
use ieee.std_logic_1164.all;
use work.conversions.all;

entity memory is
    generic(add_bits : integer := 12;
            data_bits : integer := 32);
    port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
        data_in : in std_ulogic_vector(data_bits-1 downto 0);
        data_out : out std_ulogic_vector(data_bits-1 downto 0);
        cs, mwrite : in std_ulogic;
        do_init : in std_ulogic);
    subtype word is std_ulogic_vector(data_bits-1 downto 0);
    constant nwords : integer := 2 ** add_bits;
    type ram_type is array(0 to nwords-1) of word;
end;

architecture style_93 of memory is
        ------------------------------
        shared variable ram : ram_type;
        ------------------------------
begin
memory:
process (cs)
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
    variable address : natural;
    begin
        if rising_edge(do_init) then
            for address in 0 to nwords-1 loop
                ram(address) := data_in;
            end loop;
        end if;
    end process initialize;
end architecture style_93;

architecture style_87 of memory is
begin
memory:
process (cs)
    -----------------------
    variable ram : ram_type;
    -----------------------
    variable address : natural;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                 ram(address) := data_in;
            end if;
            data_out <= ram(address);
        end if;
    end process;
end style_87;

architecture bad_style_87 of memory is
    ----------------------
    signal ram : ram_type;
    ----------------------
begin
memory:
process (cs)
    variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) <= data_in;
                data_out <= data_in;
            else
                data_out <= ram(address);
            end if;
        end if;
    end process;
end bad_style_87;

------------------------------------------------------------
------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sulv_to_natural(x : std_ulogic_vector) return
                natural;
    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector;
end conversions;

package body conversions is

    function sulv_to_natural(x : std_ulogic_vector) return
                natural is
        variable n : natural := 0;
        variable failure : boolean := false;
    begin
        assert (x'high - x'low + 1) <= 31
            report "Range of sulv_to_natural argument exceeds
                natural range"
            severity error;
        for i in x'range loop
            n := n * 2;
            case x(i) is
```

```
                when '1' | 'H' => n := n + 1;
                when '0' | 'L' => null;
                when others    => failure := true;
            end case;
        end loop;
        assert not failure
            report "sulv_to_natural cannot convert indefinite
                std_ulogic_vector"
            severity error;

        if failure then
            return 0;
        else
            return n;
        end if;
    end sulv_to_natural;

    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector is
        variable x : std_ulogic_vector(bits-1 downto 0) :=
                (others => '0');
        variable tempn : natural := n;
    begin
        for i in x'reverse_range loop
            if (tempn mod 2) = 1 then
                x(i) := '1';
            end if;
            tempn := tempn / 2;
        end loop;
        return x;
    end natural_to_sulv;

end conversions;
```

VHDL02 example

```
-------------------------------------------------------------------------
-- Source:     sp_syn_ram_protected.vhd
-- Component:  VHDL synchronous, single-port RAM
-- Remarks:    Various VHDL examples: random access memory (RAM)
-------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY sp_syn_ram_protected IS
    GENERIC (
        data_width : positive := 8;
        addr_width : positive := 3
    );
    PORT (
        inclk    : IN  std_logic;
        outclk   : IN  std_logic;
        we       : IN  std_logic;
        addr     : IN  unsigned(addr_width-1 DOWNTO 0);
        data_in  : IN  std_logic_vector(data_width-1 DOWNTO 0);
        data_out : OUT std_logic_vector(data_width-1 DOWNTO 0)
    );

END sp_syn_ram_protected;


ARCHITECTURE intarch OF sp_syn_ram_protected IS

   TYPE mem_type IS PROTECTED
     PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);
                      addr : IN unsigned(addr_width-1 DOWNTO 0));
     IMPURE FUNCTION read  (  addr : IN unsigned(addr_width-1 DOWNTO 0))
RETURN
       std_logic_vector;
    END PROTECTED mem_type;

   TYPE mem_type IS PROTECTED BODY
      TYPE mem_array IS ARRAY (0 TO 2**addr_width-1) OF
                     std_logic_vector(data_width-1 DOWNTO 0);
     VARIABLE mem : mem_array;

     PROCEDURE write ( data : IN std_logic_vector(data_width-1 downto 0);
                      addr : IN unsigned(addr_width-1 DOWNTO 0)) IS
     BEGIN
        mem(to_integer(addr)) := data;
      END;

     IMPURE FUNCTION read  ( addr : IN unsigned(addr_width-1 DOWNTO 0))
RETURN
       std_logic_vector IS
      BEGIN
         return mem(to_integer(addr));
      END;

    END PROTECTED BODY mem_type;
    SHARED VARIABLE memory : mem_type;

BEGIN

    ASSERT data_width <= 32
        REPORT "### Illegal data width detected"
        SEVERITY failure;

    control_proc : PROCESS (inclk, outclk)

    BEGIN
        IF (inclk'event AND inclk = '1') THEN
            IF (we = '1') THEN
                memory.write(data_in, addr);
            END IF;
        END IF;

        IF (outclk'event AND outclk = '1') THEN
            data_out <= memory.read(addr);
        END IF;
    END PROCESS;

END intarch;

-------------------------------------------------------------------------
-- Source:    ram_tb.vhd
-- Component: VHDL testbench for RAM memory example
-- Remarks:   Simple VHDL example: random access memory (RAM)
-------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram_tb IS
END ram_tb;

ARCHITECTURE testbench OF ram_tb IS

    -------------------------------------------
    -- Component declaration single-port RAM
    -------------------------------------------
    COMPONENT sp_syn_ram_protected
        GENERIC (
            data_width : positive := 8;
            addr_width : positive := 3
        );
        PORT (
            inclk    : IN  std_logic;
            outclk   : IN  std_logic;
            we       : IN  std_logic;
            addr     : IN  unsigned(addr_width-1 DOWNTO 0);
            data_in  : IN  std_logic_vector(data_width-1 DOWNTO 0);
            data_out : OUT std_logic_vector(data_width-1 DOWNTO 0)
        );
    END COMPONENT;

    -------------------------------------------
```

```
        -- Intermediate signals and constants
        -------------------------------------------
        SIGNAL   addr     : unsigned(19 DOWNTO 0);
        SIGNAL   inaddr   : unsigned(3 DOWNTO 0);
        SIGNAL   outaddr  : unsigned(3 DOWNTO 0);
        SIGNAL   data_in  : unsigned(31 DOWNTO 0);
        SIGNAL   data_in1 : std_logic_vector(7 DOWNTO 0);
        SIGNAL   data_sp1 : std_logic_vector(7 DOWNTO 0);
        SIGNAL   we       : std_logic;
        SIGNAL   clk      : std_logic;
        CONSTANT clk_pd   : time := 100 ns;


    BEGIN

        ---------------------------------------------------
        -- instantiations of single-port RAM architectures.
        -- All architectures behave equivalently, but they
        -- have different implementations.  The signal-based
        -- architecture (rtl) is not a recommended style.
        ---------------------------------------------------
        spram1 : entity work.sp_syn_ram_protected
            GENERIC MAP (
                data_width => 8,
                addr_width => 12)
            PORT MAP (
                inclk    => clk,
                outclk   => clk,
                we       => we,
                addr     => addr(11 downto 0),
                data_in  => data_in1,
                data_out => data_sp1);

        -------------------------------------------
        -- clock generator
        -------------------------------------------
        clock_driver : PROCESS
        BEGIN
            clk <= '0';
            WAIT FOR clk_pd / 2;
            LOOP
                clk <= '1', '0' AFTER clk_pd / 2;
                WAIT FOR clk_pd;
            END LOOP;
        END PROCESS;

        -------------------------------------------
        -- data-in process
        -------------------------------------------
        datain_drivers : PROCESS(data_in)
        BEGIN
            data_in1 <= std_logic_vector(data_in(7 downto 0));
        END PROCESS;

        -------------------------------------------
        -- simulation control process
        -------------------------------------------
        ctrl_sim : PROCESS
        BEGIN
            FOR i IN 0 TO 1023 LOOP
                we       <= '1';
                data_in  <= to_unsigned(9000 + i, data_in'length);
                addr     <= to_unsigned(i, addr'length);
                inaddr   <= to_unsigned(i, inaddr'length);
                outaddr  <= to_unsigned(i, outaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                data_in  <= to_unsigned(7 + i, data_in'length);
                addr     <= to_unsigned(1 + i, addr'length);
                inaddr   <= to_unsigned(1 + i, inaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                data_in  <= to_unsigned(3, data_in'length);
                addr     <= to_unsigned(2 + i, addr'length);
                inaddr   <= to_unsigned(2 + i, inaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                data_in  <= to_unsigned(30330, data_in'length);
                addr     <= to_unsigned(3 + i, addr'length);
                inaddr   <= to_unsigned(3 + i, inaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                we       <= '0';
                addr     <= to_unsigned(i, addr'length);
                outaddr  <= to_unsigned(i, outaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                addr     <= to_unsigned(1 + i, addr'length);
                outaddr  <= to_unsigned(1 + i, outaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                addr     <= to_unsigned(2 + i, addr'length);
                outaddr  <= to_unsigned(2 + i, outaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

                addr     <= to_unsigned(3 + i, addr'length);
                outaddr  <= to_unsigned(3 + i, outaddr'length);
                WAIT UNTIL clk'EVENT AND clk = '0';
                WAIT UNTIL clk'EVENT AND clk = '0';

            END LOOP;
            ASSERT false
                REPORT "### End of Simulation!"
                SEVERITY failure;
        END PROCESS;

    END testbench;
```

Affecting Performance by Cancelling Scheduled Events

Performance will suffer if events are scheduled far into the future but then cancelled before they take effect. This situation will act like a memory leak and slow down simulation.

In VHDL this situation can occur several ways. The most common are waits with time-out clauses and projected waveforms in signal assignments.

The following code shows a wait with a time-out:

```
signals synch : bit := '0';
...
p: process
begin
   wait for 10 ms until synch = 1;
end process;

synch <= not synch after 10 ns;
```

At time 0, process *p* makes an event for time 10ms. When *synch* goes to 1 at 10 ns, the event at 10 ms is marked as cancelled but not deleted, and a new event is scheduled at 10ms + 10ns. The cancelled events are not reclaimed until time 10ms is reached and the cancelled event is processed. As a result there will be 500000 (10ms/20ns) cancelled but un-deleted events. Once 10ms is reached, memory will no longer increase because the simulator will be reclaiming events as fast as they are added.

For projected waveforms the following would behave the same way:

```
signals synch : bit := '0';
...
p: process(synch)
begin
   output <= '0', '1' after 10ms;
end process;

synch <= not synch after 10 ns;
```

Converting an Integer Into a bit_vector

The following code demonstrates how to convert an integer into a bit_vector.

```
library ieee;
use ieee.numeric_bit.ALL;

entity test is
end test;

architecture only of test is
  signal s1 : bit_vector(7 downto 0);
  signal int : integer := 45;
begin
  p:process
  begin
    wait for 10 ns;
    s1 <= bit_vector(to_signed(int,8));
  end process p;
end only;
```

Chapter 7
Verilog and SystemVerilog Simulation

This chapter describes how to compile and simulate Verilog and SystemVerilog designs with ModelSim. ModelSim implements the Verilog language as defined by the IEEE Standards 1364-1995 and 1364-2005. We recommend that you obtain these specifications for reference.

The following functionality is partially implemented in ModelSim:

- Verilog Procedural Interface (VPI) (see */<install_dir>/modeltech/docs/technotes/Verilog_VPI.note* for details)
- IEEE Std P1800-2005 SystemVerilog (see */<install_dir>/modeltech/docs/technotes/sysvlog.note* for implementation details)

Terminology

This chapter uses the term “Verilog” to represent both Verilog and SystemVerilog, unless otherwise noted.

Basic Verilog Flow

Simulating Verilog designs with ModelSim includes four general steps:

1. Compile your Verilog code into one or more libraries using the vlog command. See Compiling Verilog Files for details.
2. Optimize your design using the vopt command. See Chapter 3, Optimizing Designs with vopt and Optimization Considerations for Verilog Designs for details.
3. Load your design with the vsim command. See Simulating Verilog Designs for details.
4. Run and debug your design.

Compiling Verilog Files

The first time you compile a design there is a two-step process:

1. Create a working library with **vlib** or select **File > New > Library**.
2. Compile the design using **vlog** or select **Compile > Compile**.

Creating a Working Library

Before you can compile your design, you must create a library in which to store the compilation results. Use the vlib command or select **File > New > Library** to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named *_info*. Do not create libraries using UNIX commands – always use the vlib.

See Design Libraries for additional information on working with libraries.

Invoking the Verilog Compiler

The Verilog compiler, **vlog**, compiles Verilog source code into retargetable, executable code. The library format is compatible across all supported platforms, and you can simulate your design on any platform without having to recompile your design.

As the design compiles, the resulting object code for modules and UDPs is generated into a library. As noted above, the compiler places results into the work library by default. You can specify an alternate library with the **-work** argument.

Example 7-1. Invocation of the Verilog Compiler

Here is a sample invocation of **vlog**:

```
vlog top.v +libext+.v+.u -y vlog_lib
```

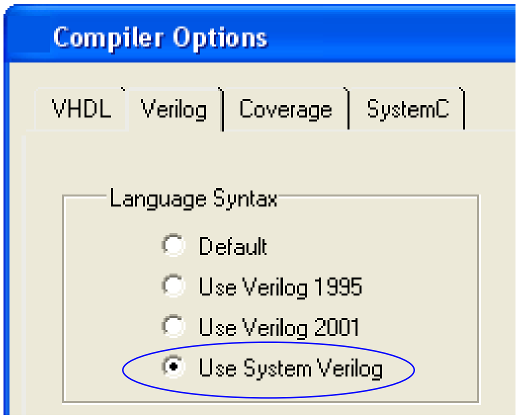
After compiling *top.v*, **vlog** will scan the *vlog_lib* library for files with modules with the same name as primitives referenced, but undefined in *top.v*. The use of **+libext+.v+.u** implies filenames with a *.v* or *.u* suffix (any combination of suffixes may be used). Only referenced definitions will be compiled.

Parsing SystemVerilog Keywords

With standard Verilog files (<*filename*>*.v*), **vlog** will not automatically parse SystemVerilog keywords. SystemVerilog keywords are parsed when any of the following situations exists:

- any file within the design contains the *.sv* file extension,
- the **-sv** argument is used with the vlog command,

- the Use System Verilog option is selected in the Verilog tab of the Compiler Options dialog. Access this dialog by selecting **Compile > Compile Options** from the Main window menu bar.



Language Syntax

○ Default
○ Use Verilog 1995
○ Use Verilog 2001
● Use System Verilog

Here are two examples of the **vlog** command that will enable SystemVerilog features and keywords in ModelSim:

```
vlog testbench.sv top.v memory.v cache.v
vlog -sv testbench.v proc.v
```

In the first example, the *.sv* extension for *testbench* automatically instructs ModelSim to parse SystemVerilog keywords. The *-sv* option used in the second example enables SystemVerilog features and keywords.

Though a primary goal of the SystemVerilog standardization efforts has been to ensure full backward compatibility with the Verilog standard, there is an issue with keywords. SystemVerilog adds several new keywords to the Verilog language (see Table B-1 in Appendix B of the P1800 SystemVerilog standard). If your design uses one of these keywords as a regular identifier for a variable, module, task, function, etc., your design will not compile in ModelSim.

Incremental Compilation

ModelSim Verilog supports incremental compilation of designs. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler.

You are not required to compile your design in any particular order (unless you are using SystemVerilog packages; see note below) because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator.

Note

Compilation order may matter when using SystemVerilog packages. As stated in the IEEE std p1800-2005 LRM, section entitled *Referencing data in packages*, which states: "Packages must exist in order for the items they define to be recognized by the scopes in which they are imported.”

Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

Example 7-2. Incremental Compilation Example

Contents of testbench.sv

```
module testbench;
   timeunit 1ns;
   timeprecision 10ps;
   bit d=1, clk = 0;
   wire q;
   initial
      for (int cycles=0; cycles < 100; cycles++)
         #100 clk = !clk;

   design dut(q, d, clk);
endmodule
```

Contents of design.v:

```
module design(output bit q, input bit d, clk);
   timeunit 1ns;
   timeprecision 10ps;
   always @(posedge clk)
      q = d;
endmodule
```

Compile the design incrementally as follows:

```
ModelSim> vlog testbench.sv
.
# Top level modules:
#       testbench
ModelSim> vlog -sv test1.v
.
# Top level modules:
#       dut
```

Note that the compiler lists each module as a top-level module, although, ultimately, only *testbench* is a top-level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top-level module. This is just an informative message and can be ignored during incremental compilation.

The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top-level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
        top
```

Automatic Incremental Compilation with -incr

The most efficient method of incremental compilation is to manually compile only the modules that have changed. However, this is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to compile your entire design along with the **-incr** argument. This causes the compiler to automatically determine which modules have changed and generate code only for those modules.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
        top
```

Now, suppose that you modify the functionality of the *or2* module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
        top
```

The compiler informs you that it skipped the modules *top* and *and2*, and compiled *or2*.

Automatic incremental compilation is intelligent about when to compile a module. For example, changing a comment in your source code does not result in a recompile; however, changing the compiler command line arguments results in a recompile of all modules.

Note

Changes to your source code that do not change functionality but that do affect source code line numbers (such as adding a comment line) *will* cause all affected modules to be recompiled. This happens because debug information must be kept current so that ModelSim can trace back to the correct areas of the source code.

Library Usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2

Top level modules:
        and2
        or2
% vlog top.v
-- Compiling module top

Top level modules:
        top
```

Note that the first compilation uses the **-work asiclib** argument to instruct the compiler to place the results in the **asiclib** library rather than the default **work** library.

Library Search Rules for vlog

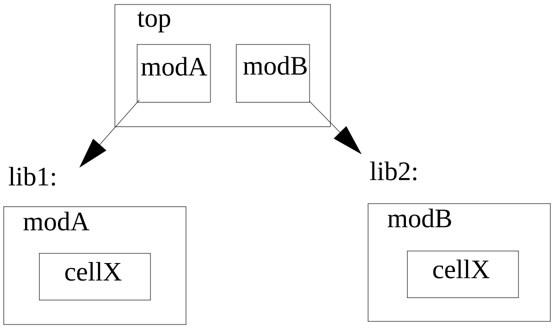
Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top-level modules are loaded from the library named **work** unless you prefix the modules with the **<library>.** option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** arguments in the order they appear on the command line.
- Search the library specified in the Verilog-XL uselib Compiler Directive section.
- Search libraries specified with **-L** arguments in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

Handling Sub-Modules with Common Names

Sometimes in one design you need to reference two different modules that have the same name. This situation can occur if you have hierarchical modules organized into separate libraries, and you have commonly-named sub-modules in the libraries that have different definitions. This may happen if you are using vendor-supplied libraries.

For example, say you have the following:



The normal library search rules will fail in this situation. For example, if you load the design as follows:

```
vsim -L lib1 -L lib2 top
```

both instantiations of *cellX* resolve to the *lib1* version of *cellX*. On the other hand, if you specify *-L lib2 -L lib1*, both instantiations of *cellX* resolve to the *lib2* version of *cellX*.

To handle this situation, ModelSim implements a special interpretation of the expression *-L work*. When you specify *-L work* first in the search library arguments you are directing **vsim** to search for the instantiated module or UDP in the library that contains the module that does the instantiation.

In the example above you would invoke vsim as follows:

```
vsim -L work -L lib1 -L lib2 top
```

SystemVerilog Multi-File Compilation Issues

Declarations in Compilation Unit Scope

SystemVerilog allows the declaration of types, variables, functions, tasks, and other constructs in compilation unit scope ($unit). The visibility of declarations in **$unit** scope does not extend outside the current compilation unit. Thus, it is important to understand how compilation units are defined by the tool during compilation.

By default, vlog operates in Single File Compilation Unit mode (SFCU). This means the visibility of declarations in **$unit** scope terminates at the end of each source file. Visibility does not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the declaration to the file containing the end of the declaration.

vlog also supports a non-default behavior called Multi File Compilation Unit mode (MFCU). In MFCU mode, **vlog** compiles all files given on the command line into one compilation unit. You can invoke **vlog** in MFCU mode as follows:

- For a specific compilation -- with the **-mfcu** argument to vlog.
- For all compilations -- by setting the variable **MultiFileCompilationUnit = 1** in the *modelsim.ini* file.

By using either of these methods, you allow declarations in **$unit** scope to remain in effect throughout the compilation of all files.

In case you have made MFCU the default behavior by setting **MultiFileCompilationUnit = 1** in your modelsim.ini file, it is possible to override the default behavior on specific compilations by using the **-sfcu** argument to vlog.

Macro Definitions and Compiler Directives in Compilation Unit Scope

According to the SystemVerilog IEEE Std p1800-2005 LRM, the visibility of macro definitions and compiler directives span the lifetime of a single compilation unit. By default, this means the definitions of macros and settings of compiler directives terminate at the end of each source file. They do not carry forward from one file to another, except when a module, interface, or package declaration begins in one file and ends in another file. In that case, the compilation unit spans from the file containing the beginning of the definition to the file containing the end of the definition.

See Declarations in Compilation Unit Scope for instructions on how to control vlog's handling of compilation units.

Note

Compiler directives revert to their default values at the end of a compilation unit.

If a compiler directive is specified as an option to the compiler, this setting is used for all compilation units present in the current compilation.

Verilog-XL Compatible Compiler Arguments

The compiler arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the vlog command for a description of each argument.

```
+define+<macro_name>[=<macro_text>]
+delay_mode_distributed
+delay_mode_path
+delay_mode_unit
+delay_mode_zero
-f <filename>
+incdir+<directory>
+mindelays
+maxdelays
+nowarn<mnemonic>
+typdelays
-u
```

Arguments Supporting Source Libraries

The compiler arguments listed below support source libraries in the same manner as Verilog-XL. See the vlog command for a description of each argument.

Note that these source libraries are very different from the libraries that the ModelSim compiler uses to store compilation results. You may find it convenient to use these arguments if you are porting a design to ModelSim or if you are familiar with these arguments and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

```
-v <filename>
-y <directory>
+libext+<suffix>
+librescan
+nolibcell
-R [<simargs>]
```

Verilog-XL uselib Compiler Directive

The **`uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler arguments. It has the advantage that a design may reference different modules having the same name. You compile designs that contain **`uselib** directive statements using the **-compile_uselibs** argument (described below) to vlog.

The syntax for the **`uselib** directive is:

```
`uselib <library_reference>...
```

where <library_reference> can be one or more of the following:

- **dir=<library_directory>**, which is equivalent to the command line argument:

  ```
  -y <library_directory>
  ```

- **file=<library_file>**, which is equivalent to the command line argument:

  ```
  -v <library_file>
  ```

- **libext=<file_extension>**, which is equivalent to the command line argument:

  ```
  +libext+<file_extension>
  ```

- **lib=<library_name>**, which references a library for instantiated objects. This behaves similarly to a LIBRARY/USE clause in VHDL. You must ensure the correct mappings are set up if the library does not exist in the current working directory. The **-compile_uselibs** argument does not affect this usage of **`uselib**.

For example, the following directive

```
`uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line arguments:

```
-y /h/vendorA +libext+.v
```

Since the **`uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **`uselib** directives.

-compile_uselibs Argument

Use the **-compile_uselibs** argument to vlog to reference **`uselib** directives. The argument finds the source files referenced in the directive, compiles them into automatically created object libraries, and updates the *modelsim.ini* file with the logical mappings to the libraries.

When using **-compile_uselibs**, ModelSim determines into which directory to compile the object libraries by choosing, in order, from the following three values:

- The directory name specified by the **-compile_uselibs** argument. For example,

  ```
  -compile_uselibs=./mydir
  ```

- The directory specified by the MTI_USELIB_DIR environment variable (see Environment Variables)

- A directory named *mti_uselibs* that is created in the current working directory

The following code fragment and compiler invocation show how two different modules that have the same name can be instantiated within the same design:

```
module top;
  `uselib dir=/h/vendorA libext=.v
  NAND2 u1(n1, n2, n3);
  `uselib dir=/h/vendorB libext=.v
  NAND2 u2(n4, n5, n6);
endmodule
```

vlog -compile_uselibs top

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

uselib is Persistent

As mentioned above, the appearance of a **`uselib** directive in the source code explicitly defines how instantiations that follow it are resolved. This may result in unexpected consequences. For example, consider the following compile command:

vlog -compile_uselibs dut.v srtr.v

Assume that *dut.v* contains a **`uselib** directive. Since *srtr.v* is compiled after *dut.v*, the **`uselib** directive is still in effect. When *srtr* is loaded it is using the **`uselib** directive from *dut.v* to decide where to locate modules. If this is not what you intend, then you need to put an empty **`uselib** at the end of *dut.v* to "close" the previous **`uselib** statement.

Verilog Configurations

The Verilog 2001 specification added configurations. Configurations specify how a design is "assembled" during the elaboration phase of simulation. Configurations actually consist of two pieces: the library mapping and the configuration itself. The library mapping is used at compile time to determine into which libraries the source files are to be compiled. Here is an example of a simple library map file:

```
library work    ../top.v;
library rtlLib  lrm_ex_top.v;
library gateLib lrm_ex_adder.vg;
library aLib    lrm_ex_adder.v;
```

Here is an example of a library map file that uses **-incdir**:

```
library lib1 src_dir/*.v -incdir ../include_dir2, ../, my_incdir;
```

The name of the library map file is arbitrary. You specify the library map file using the **-libmap** argument to the vlog command. Alternatively, you can specify the file name as the first item on the **vlog** command line, and the compiler will read it as a library map file.

The library map file must be compiled along with the Verilog source files. Multiple map files are allowed but each must be preceded by the **-libmap** argument.

The library map file and the configuration can exist in the same or different files. If they are separate, only the map file needs the **-libmap** argument. The configuration is treated as any other Verilog source file.

Configurations and the Library Named work

The library named "work" is treated specially by ModelSim (see The Library Named "work" for details) for Verilog configurations. Consider the following code example:

```
config cfg;
  design top;
  instance top.u1 use work.u1;
endconfig
```

In this case, *work.u1* indicates to load *u1* from the current library.

Verilog Generate Statements

The Verilog 2001 rules for generate statements had numerous inconsistencies and ambiguities. As a result, ModelSim implements the rules that have been adopted for Verilog 2005. Most of the rules are backwards compatible, but there is one key difference related to name visibility.

Name Visibility in Generate Statements

Consider the following code example:

```
module m;
   parameter p = 1;

   generate
   if (p)
      integer x = 1;
   else
      real x = 2.0;
   endgenerate

   initial $display(x);
endmodule
```

This code sample is legal under 2001 rules. However, it is illegal under the 2005 rules and will cause an error in ModelSim. Under the new rules, you cannot hierarchically reference a name in an anonymous scope from outside that scope. In the example above, *x* does not propagate its visibility upwards, and each condition alternative is considered to be an anonymous scope.

To fix the code such that it will simulate properly in ModelSim, write it like this instead:

```
module m;
   parameter p = 1;

   if (p) begin:s
      integer x = 1;
   end
   else begin:s
      real x = 2.0;
   end

   initial $display(s.x);
endmodule
```

Since the scope is named in this example, normal hierarchical resolution rules apply and the code is fine.

Note too that the keywords **generate - endgenerate** are optional under the 2005 rules and are excluded in the second example.

Simulating Verilog Designs

A Verilog design is ready for simulation after it has been compiled with **vlog** and possibly optimized with **vopt**. For more information on Verilog optimizations, see Chapter 3, Optimizing Designs with vopt and Optimization Considerations for Verilog Designs. The simulator may then be invoked with the names of the top-level modules (many designs contain only one top-level module) or the name you assigned to the optimized version of the design. For example, if your top-level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

After the simulator loads the top-level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting the ports and resolving hierarchical references. By default all modules and UDPs are loaded from the library named **work**. Modules and UDPs from other libraries can be specified using the **-L** or **-Lf** arguments to **vsim** (see Library Usage for details).

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **$finish** is executed in the Verilog code. You can also run for specific time periods (e.g., run 100 ns). Enter the **quit** command to exit the simulator.

Simulator Resolution Limit (Verilog)

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulator resolution limit. The resolution limit defaults to the smallest time precision found among all of the **`timescale** compiler directives in the design. Here is an example of a **`timescale** directive:

```
`timescale 1 ns / 100 ps
```

The first number is the time units and the second number is the time precision. The directive above causes time values to be read as ns and to be rounded to the nearest 100 ps.

Time units and precision can also be specified with SystemVerilog keywords as follows:

```
timeunit 1 ns
timeprecision 100 ps
```

Modules Without Timescale Directives

You may encounter unexpected behavior if your design contains some modules with timescale directives and others without. The time units for modules without a timescale directive default to the simulator resolution. For example, say you have the two modules shown in the table below:

Table 7-1.

Module 1	Module 2

`timescale 1 ns / 10 ps
 	
module mod1 (set);
 	
  output set;
  reg set;
  parameter d = 1.55;
 	
  initial
  begin
    set = 1'bz;
    #d set = 1'b0;
    #d set = 1'b1;
  end
 	
endmodule
	module mod2 (set);
	
	  output set;
	  reg set;
	  parameter d = 1.55;
	
	  initial
	  begin
	    set = 1'bz;
	    #d set = 1'b0;
	    #d set = 1'b1;
	  end
	
	endmodule

If you invoke **vsim** as *vsim mod2 mod1* then Module 1 sets the simulator resolution to 10 ps. Module 2 has no timescale directive, so the time units default to the simulator resolution, in this case 10 ps. If you watched */mod1/set* and */mod2/set* in the Wave window, you'd see that in Module 1 it transitions every 1.55 ns as expected (because of the 1 ns time unit in the timescale directive). However, in Module 2, *set* transitions every 20 ps. That's because the delay of 1.55 in Module 2 is read as 15.5 ps and is rounded up to 20 ps.

In such cases ModelSim will issue the following warning message during elaboration:

```
** Warning: (vsim-3010) [TSCALE] - Module 'mod1' has a `timescale
directive in effect, but previous modules do not.
```

If you invoke **vsim** as vsim mod1 mod2, the simulation results would be the same but ModelSim would produce a different warning message:

```
** Warning: (vsim-3009) [TSCALE] - Module 'mod2' does not have a
`timescale directive in effect, but previous modules do.
```

These warnings should ALWAYS be investigated.

If the design contains no `timescale directives, then the resolution limit and time units default to the value specified by the Resolution variable in the *modelsim.ini* file. (The variable is set to 1 ns by default.)

-timescale Option

The **-timescale** option can be used with the **vlog** and **vopt** to specifies the default timescale for modules not having an explicit **`timescale** directive in effect during compilation. The format of the **-timescale** argument is the same as that of the **`timescale** directive

```
-timescale <time_units>/<time_precision>
```

The format for <*time_units*> and <*time_precision*> is <*n*><*units*>. The value of <*n*> must be 1, 10, or 100. The value of <*units*> must be fs, ps, ns, us, ms, or s. In addition, the <*time_units*> must be greater than or equal to the <*time_precision*>. For example:

```
-timescale "1ns / 1ps"
```

The argument above needs quotes because it contains white space.

Multiple Timescale Directives

As alluded to above, your design can have multiple timescale directives. The timescale directive takes effect where it appears in a source file and applies to all source files which follow in the same vlog command. Separately compiled modules can also have different timescales. The simulator determines the smallest timescale of all the modules in a design and uses that as the simulator resolution.

timescale, -t, and Rounding

The optional **vsim** argument **-t** sets the simulator resolution limit for the overall simulation. If the resolution set by **-t** is larger than the precision set in a module, the time values in that module are rounded up. If the resolution set by **-t** is smaller than the precision of the module, the precision of that module remains whatever is specified by the `timescale directive. Consider the following code:

```
`timescale 1 ns / 100 ps

module foo;

  initial
    #12.536 $display
```

The list below shows three possibilities for **-t** and how the delays in the module would be handled in each case:

- **-t** not set

  The delay will be rounded to 12.5 as directed by the module's ‘timescale directive.

- **-t** is set to 1 fs

  The delay will be rounded to 12.5. Again, the module's precision is determined by the ‘timescale directive. ModelSim does not override the module's precision.

- **-t** is set to 1 ns

  The delay will be rounded to 12. The module's precision is determined by the **-t** setting. ModelSim has no choice but to round the module's time values because the entire simulation is operating at 1 ns.

Choosing the Resolution for Verilog

You should choose the coarsest resolution limit possible that does not result in undesired rounding of your delays. The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases.

Event Ordering in Verilog Designs

Event-based simulators such as ModelSim may process multiple events at a given simulation time. The Verilog language is defined such that you cannot explicitly control the order in which simultaneous events are processed. Unfortunately, some designs rely on a particular event order, and these designs may behave differently than you expect.

Event Queues

Section 5 of the IEEE Std 1364-1995 LRM defines several event queues that determine the order in which events are evaluated. At the current simulation time, the simulator has the following pending events:

- active events
- inactive events
- non-blocking assignment update events

- monitor events
- future events
    - inactive events
    - non-blocking assignment update events

The LRM dictates that events are processed as follows – 1) all active events are processed; 2) the inactive events are moved to the active event queue and then processed; 3) the non-blocking events are moved to the active event queue and then processed; 4) the monitor events are moved to the active queue and then processed; 5) simulation advances to the next time where there is an inactive event or a non-blocking assignment update event.

Within the active event queue, the events can be processed in any order, and new active events can be added to the queue in any order. In other words, you *cannot* control event order within the active queue. The example below illustrates potential ramifications of this situation.

Say you have these four statements:

1. always@(q) p = q;
2. always @(q) p2 = not q;
3. always @(p or p2) clk = p and p2;
4. always @(posedge clk)

and current values as follows: q = 0, p = 0, p2=1

The tables below show two of the many valid evaluations of these statements. Evaluation events are denoted by a number where the number is the statement to be evaluated. Update events are denoted *<name>(old->new)* where *<name>* indicates the reg being updated and *new* is the updated value.\

Table 7-2.

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
p(0 -> 1)	3, 2
3	clk(0 -> 1), 2
clk(0 -> 1)	4, 2
4	2
2	p2(1 -> 0)

Table 7-2.

Event being processed	Active event queue
p2(1 -> 0)	3
3	clk(1 -> 0)
clk(1 -> 0)	<empty>

Table 7-3.

Event being processed	Active event queue
	q(0 -> 1)
q(0 -> 1)	1, 2
1	p(0 -> 1), 2
2	p2(1 -> 0), p(0 -> 1)
p(0 -> 1)	3, p2(1 -> 0)
p2(1 –> 0)	3
3	<empty> (clk doesn’t change)

Again, both evaluations are valid. However, in Evaluation 1, *clk* has a glitch on it; in Evaluation 2, *clk* doesn’t. This indicates that the design has a zero-delay race condition on *clk*.

Controlling Event Queues with Blocking or Non-Blocking Assignments

The only control you have over event order is to assign an event to a particular queue. You do this via blocking or non-blocking assignments.

Blocking Assignments

Blocking assignments place an event in the active, inactive, or future queues depending on what type of delay they have:

- a blocking assignment without a delay goes in the active queue

- a blocking assignment with an explicit delay of 0 goes in the inactive queue
- a blocking assignment with a non-zero delay goes in the future queue

Non-Blocking Assignments

A non-blocking assignment goes into either the non-blocking assignment update event queue or the future non-blocking assignment update event queue. (Non-blocking assignments with no delays and those with explicit zero delays are treated the same.)

Non-blocking assignments should be used only for outputs of flip-flops. This insures that all outputs of flip-flops do not change until after all flip-flops have been evaluated. Attempting to use non-blocking assignments in combinational logic paths to remove race conditions may only cause more problems. (In the preceding example, changing all statements to non-blocking assignments would not remove the race condition.) This includes using non-blocking assignments in the generation of gated clocks.

The following is an example of how to properly use non-blocking assignments.

```
gen1: always @(master)
  clk1 = master;

gen2: always @(clk1)
  clk2 = clk1;

f1 : always @(posedge clk1)
  begin
    q1 <= d1;
  end

f2:   always @(posedge clk2)
  begin
    q2 <= q1;
  end
```

If written this way, a value on *d1* always takes two clock cycles to get from *d1* to *q2*. If you change *clk1 = master* and *clk2 = clk1* to non-blocking assignments or *q2 <= q1* and *q1 <= d1* to blocking assignments, then *d1* may get to *q2* is less than two clock cycles.

Debugging Event Order Issues

Since many models have been developed on Verilog-XL, ModelSim tries to duplicate Verilog-XL event ordering to ease the porting of those models to ModelSim. However, ModelSim does not match Verilog-XL event ordering in all cases, and if a model ported to ModelSim does not behave as expected, then you should suspect that there are event order dependencies.

ModelSim helps you track down event order dependencies with the following compiler arguments: **-compat**, **-hazards**, and **-keep_delta**.

See the vlog command for descriptions of **-compat** and **-hazards**.

Hazard Detection

The **-hazard** argument to vsim detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. **vsim** detects the following kinds of hazards:

- WRITE/WRITE — Two processes writing to the same variable at the same time.
- READ/WRITE — One process reading a variable at the same time it is being written to by another process. ModelSim calls this a READ/WRITE hazard if it executed the read first.
- WRITE/READ — Same as a READ/WRITE hazard except that ModelSim executed the write first.

vsim issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **Error**.

To enable hazard detection you must invoke vlog with the **-hazards** argument when you compile your source code and you must also invoke **vsim** with the **-hazards** argument when you simulate.

Note

Enabling **-hazards** implicitly enables the **-compat** argument. As a result, using this argument may affect your simulation results.

Hazard Detection and Optimization Levels

In certain cases hazard detection results are affected by the optimization level used in the simulation. Some optimizations change the read/write operations performed on a variable if the transformation is determined to yield equivalent results. Since the hazard detection algorithm doesn’t know whether or not the read/write operations can affect the simulation results, the optimizations can result in different hazard detection results. Generally, the optimizations reduce the number of false hazards by eliminating unnecessary reads and writes, but there are also optimizations that can produce additional false hazards.

Limitations of Hazard Detection

- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.
- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.

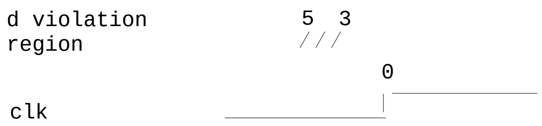
- Glitches on nets caused by non-guaranteed event ordering are not detected.
- A non-blocking assignment is not treated as a WRITE for hazard detection purposes. This is because non-blocking assignments are not normally involved in hazards. (In fact, they should be used to avoid hazards.)

Negative Timing Check Limits

Verilog supports negative limit values in the $setuphold and $recrem system tasks. These tasks have optional delayed versions of input signals to insure proper evaluation of models with negative timing check limits. Delay values for these delayed nets are determined by the simulator so that valid data is available for evaluation before a clocking signal.

Example 7-3. Negative Timing Check

```
$setuphold(posedge clk, negedge d, 5, -3, Notifier,,, clk_dly, d_dly);
```



ModelSim calculates the delay for signal *d_dly* as 4 time units instead of 3. It does this to prevent *d_dly* and *clk_dly* from occurring simultaneously when a violation isn't reported.

ModelSim accepts negative limit checks by default, unlike current versions of Verilog-XL. To match Verilog-XL default behavior (i.e., zeroing all negative timing check limits), use the **+no_neg_tcheck** argument to vsim.

Negative Timing Constraint Algorithm

The algorithm ModelSim uses to calculate delays for delayed nets isn't described in IEEE Std 1364. Rather, ModelSim matches Verilog-XL behavior. The algorithm attempts to find a set of delays so the data net is valid when the clock net transitions and the timing checks are satisfied. The algorithm is iterative because a set of delays can be selected that satisfies all timing checks for a pair of inputs but then causes mis-ordering of another pair (where both pairs of inputs share a common input). When a set of delays that satisfies all timing checks is found, the delays are said to converge.

Verilog-XL Compatible Simulator Arguments

The simulator arguments listed below are equivalent to Verilog-XL arguments and may ease the porting of a design to ModelSim. See the vsim command for a description of each argument.

```
+alt_path_delays
-l <filename>
+maxdelays
+mindelays
+multisource_int_delays
+no_cancelled_e_msg
+no_neg_tchk
+no_notifier
+no_path_edge
+no_pulse_msg
-no_risefall_delaynets
+no_show_cancelled_e
+nosdfwarn
+nowarn<mnemonic>
+ntc_warn
+pulse_e/<percent>
+pulse_e_style_ondetect
+pulse_e_style_onevent
+pulse_int_e/<percent>
+pulse_int_r/<percent>
+pulse_r/<percent>
+sdf_nocheck_celltype
+sdf_verbose
+show_cancelled_e
+transport_int_delays
+transport_path_delays
+typdelays
```

Simulating Verilog with an Elaboration File

Overview

The ModelSim compiler generates a library format that is compatible across platforms. This means the simulator can load your design on any supported platform without having to recompile first. Though this architecture offers a benefit, it also comes with a possible detriment: the simulator has to generate platform-specific code every time you load your design. This impacts the speed with which the design is loaded.

You can generate a loadable image (elaboration file) which can be simulated repeatedly. On subsequent simulations, you load the elaboration file rather than loading the design "from scratch." Elaboration files load quickly.

Why an Elaboration File?

In many cases design loading time is not that important. For example, if you’re doing "iterative design," where you simulate the design, modify the source, recompile and resimulate, the load time is just a small part of the overall flow. However, if your design is locked down and only the test vectors are modified between runs, loading time may materially impact overall simulation time, particularly for large designs loading SDF files.

Another reason to use elaboration files is for benchmarking purposes. Other simulator vendors use elaboration files, and they distinguish between elaboration and run times. If you are benchmarking ModelSim against another simulator that uses elaboration, make sure you use an elaboration file with ModelSim as well so you're comparing like to like.

One caveat with elaboration files is that they must be created and used in the same environment. The same environment means the same hardware platform, the same OS and patch version, and the same version of any PLI/FLI code loaded in the simulation.

Elaboration File Flow

We recommend the following flow to maximize the benefit of simulating elaboration files.

1. If timing for your design is fixed, include all timing data when you create the elaboration file (using the **-sdf<type> instance=<filename>** argument). If your timing is not fixed in a Verilog design, you'll have to use $sdf_annotate system tasks. Note that use of $sdf_annotate causes timing to be applied after elaboration.
2. Apply all normal vsim arguments when you create the elaboration file. Some arguments (primarily related to stimulus) may be superseded later during loading of the elaboration file (see Modifying Stimulus below).
3. Load the elaboration file along with any arguments that modify the stimulus (see below).

Creating an Elaboration File

Elaboration file creation is performed with the same **vsim** settings or switches as a normal simulation *plus* an elaboration specific argument. The simulation settings are stored in the elaboration file and dictate subsequent simulation behavior. Some of these simulation settings can be modified at elaboration file load time, as detailed below.

To create an elaboration file, use the **-elab <filename>** or **-elab_cont <filename>** argument to vsim.

The **-elab_cont** argument is used to create the elaboration file then continue with the simulation after the elaboration file is created. You can use the **-c** switch with **-elab_cont** to continue the simulation in command-line mode.

Note

Elaboration files can be created in command-line mode *only*. You cannot create an elaboration file while running the ModelSim GUI.

Loading an Elaboration File

To load an elaboration file, use the **-load_elab <filename>** argument to vsim. By default the elaboration file will load in command-line mode or interactive mode depending on the argument (-c or -i) used during elaboration file creation. If no argument was used during creation, the -load_elab argument will default to the interactive mode.

The **vsim** arguments listed below can be used with **-load_elab** to affect the simulation.

```
+<plus_args>
-c or -i
-do <do_file>
-vcdread <filename>
-vcdstim <filename>
-filemap_elab <HDLfilename>=<NEWfilename>
-l <log_file>
-trace_foreign <level>
-quiet
-wlf <filename>
```

Modification of an argument that was specified at elaboration file creation, in most cases, causes the previous value to be replaced with the new value. Usage of the **-quiet** argument at elaboration load causes the mode to be toggled from its elaboration creation setting.

All other vsim arguments must be specified when you create the elaboration file, and they cannot be used when you load the elaboration file.

Note

The elaboration file must be loaded under the same environment in which it was created. The same environment means the same hardware platform, the same OS and patch version, the same version of any PLI/FLI code loaded in the simulation, and and the same release of ModelSim.

Modifying Stimulus

A primary use of elaboration files is repeatedly simulating the same design with different stimulus. The following mechanisms allow you to modify stimulus for each run.

- Use of the change command to modify parameters or generic values. This affects values only; it has no effect on triggers, compiler directives, or generate statements that reference either a generic or parameter.
- Use of the **-filemap_elab <HDLfilename>=<NEWfilename>** argument to establish a map between files named in the elaboration file. The **<HDLfilename>** file name, if it appears in the design as a file name (for example, a VHDL FILE object as well as some Verilog sysfuncs that take file names), is substituted with the **<NEWfilename>** file name. This mapping occurs before environment variable expansion and can’t be used to redirect stdin/stdout.

- VCD stimulus files can be specified when you load the elaboration file. Both vcdread and vcdstim are supported. Specifying a different VCD file when you load the elaboration file supersedes a stimulus file you specify when you create the elaboration file.
- In Verilog, the use of **+args** which are readable by the PLI routine **mc_scan_plusargs()**. **+args** values specified when you create the elaboration file are superseded by **+args** values specified when you load the elaboration file.

Using With the PLI or FLI

PLI models do not require special code to function with an elaboration file as long as the model doesn't create simulation objects in its standard tf routines. The sizetf, misctf and checktf calls that occur during elaboration are played back at **-load_elab** to ensure the PLI model is in the correct simulation state. Registered user tf routines called from the Verilog HDL will not occur until **-load_elab** is complete and the PLI model's state is restored.

By default, FLI models are activated for checkpoint during elaboration file creation and are activated for restore during elaboration file load. (See the "Using checkpoint/restore with the FLI" section of the Foreign Language Interface Reference manual for more information.) FLI models that support checkpoint/restore will function correctly with elaboration files.

FLI models that don't support checkpoint/restore may work if simulated with the **-elab_defer_fli** argument. When used in tandem with **-elab**, **-elab_defer_fli** defers calls to the FLI model's initialization function until elaboration file load time. Deferring FLI initialization skips the FLI checkpoint/restore activity (callbacks, mti_IsRestore(), ...) and may allow these models to simulate correctly. However, deferring FLI initialization also causes FLI models in the design to be initialized in order with the entire design loaded. FLI models that are sensitive to this ordering may still not work correctly even if you use **-elab_defer_fli**.

Syntax

See the vsim command for details on **-elab**, **-elab_cont**, **-elab_defer_fli**, **-compress_elab**, **-filemap_elab**, and **-load_elab**.

Example

Upon first simulating the design, use **vsim -elab <filename> <library_name.design_unit>** to create an elaboration file that will be used in subsequent simulations.

In subsequent simulations you simply load the elaboration file (rather than the design) with **vsim -load_elab <filename>**.

To change the stimulus without recoding, recompiling, and reloading the entire design, ModelSim allows you to map the stimulus file (or files) of the original design unit to an alternate file (or files) with the **-filemap_elab** switch. For example, the VHDL code for initiating stimulus might be:

```
FILE vector_file : text IS IN "vectors";
```

where *vectors* is the stimulus file.

If the alternate stimulus file is named, say, *alt_vectors*, then the correct syntax for changing the stimulus without recoding, recompiling, and reloading the entire design is as follows:

```
vsim -load_elab <filename> -filemap_elab vectors=alt_vectors
```

Checkpointing and Restoring Verilog Simulations

The checkpoint and restore commands allow you to save and restore the simulation state within the same invocation of **vsim** or between **vsim** sessions.

Table 7-4.

Action	Definition	Command used
checkpoint	saves the simulation state	checkpoint <filename>
"warm" restore	restores a checkpoint file saved in a current **vsim** session	restore <filename>
"cold" restore	restores a checkpoint file saved in a previous invocation of **vsim**	vsim -restore <filename>

Checkpoint File Contents

The following things are saved with **checkpoint** and restored with the **restore** command:

- simulation kernel state
- *vsim.wlf* file
- signals listed in the List and Wave windows
- file pointer positions for files opened under VHDL
- file pointer positions for files opened by the Verilog **$fopen** system task
- state of foreign architectures
- state of PLI/VPI/DPI code

Checkpoint Exclusions

You *cannot* checkpoint/restore the following:

- state of macros
- changes made with the command-line interface (such as user-defined Tcl commands)
- state of graphical user interface windows
- toggle statistics

If you use the foreign interface, you will need to add additional function calls in order to use **checkpoint/restore**. See the Foreign Language Interface Reference Manual or Verilog PLI / VPI / DPI for more information.

Controlling Checkpoint File Compression

The checkpoint file is normally compressed. To turn off the compression, use the following command:

```
set CheckpointCompressMode 0
```

To turn compression back on, use this command:

```
set CheckpointCompressMode 1
```

You can also control checkpoint compression using the *modelsim.ini* file in the [vsim] section (use the same 0 or 1 switch):

```
[vsim]
CheckpointCompressMode = <switch>
```

The Difference Between Checkpoint/restore and Restart

The restart command resets the simulator to time zero, clears out any logged waveforms, and closes any files opened under VHDL and the Verilog $fopen system task. You can get the same effect by first doing a checkpoint at time zero and later doing a restore. Using **restart,** however, is likely to be faster and you don't have to save the checkpoint. To set the simulation state to anything other than time zero, you need to use **checkpoint/restore**.

Using Macros with Restart and Checkpoint/restore

The restart command resets and restarts the simulation kernel, and zeros out any user-defined commands, but it does not touch the state of the macro interpreter. This lets you do **restart** commands within macros.

The pause mode indicates that a macro has been interrupted. That condition will not be affected by a restart, and if the restart is done with an interrupted macro, the macro will still be interrupted after the restart.

The situation is similar for using **checkpoint/restore** without quitting ModelSim; that is, doing a checkpoint and later in the same session doing a restore of the earlier checkpoint. The **restore** does not touch the state of the macro interpreter so you may also do **checkpoint** and **restore** commands within macros.

Checkpointing Foreign C Code That Works with Heap Memory

If checkpointing foreign C code (FLI/PLI/VPI/DPI) that works with heap memory, use mti_Malloc() rather than raw malloc() or new. Any memory allocated with mti_Malloc() is guaranteed to be restored correctly. Any memory allocated with raw malloc() will not be restored correctly, and simulator crashes can result.

Checkpointing a Running Simulation

In general you can invoke a checkpoint command only when the simulation is stopped. If you need to checkpoint without stopping the simulation, you need to write a script that utilizes the when command and variables from your code to trigger a checkpoint. The example below show how this might be done with a simple Verilog design.

Keep in mind that the variable(s) in your code must be visible at the simulation time that the checkpoint will occur. Some global optimizations performedby ModelSim may limit variable visibility, and you may need to optimize your design using the **+acc** argument to vopt.

You would compile and run the example like this:

```
vlog when.v
vsim -c when -do "do when.do"
```

where *when.do* is:

```
onbreak {
    echo "Resume macro at $now"
    resume
}
quietly set continueSim 1
quietly set whenFired 0
quietly set checkpointCntr 0

when { needToSave = 1 } {
    echo "when Stopping to allow checkpoint at $now"
    set whenFired 1
    stop
}

while {$continueSim} {
    run -all
    if { $whenFired} {
       set whenFired 0
       echo "Out of run command. Do checkpoint here"
       checkpoint cpf.n[incr checkpointCntr].cpt
    }
}
```

and *when.v* is:

```
module when;

   reg clk;
   reg [3:0] cnt;
   reg  needToSave;

   initial
     begin
        needToSave = 0;
        clk = 0;
       cnt = 0;
       #1000;
       $display("Done at time %t", $time);
       $finish;
    end

   always #10 clk = ~clk;

   always @(posedge clk)
     begin
     cnt = cnt + 1;
     if (cnt == 4'hF)
       begin
          $display( "Need to Save : %b", needToSave);
          needToSave = 1;
       end
     end

   // Need to reset the flag, but must wait a timestep
   // so that the when command has a chance to fire

   always @(posedge needToSave)
     #1 needToSave = 0;
endmodule
```

and the transcript output is:

```
# vsim -do {do when.do} -c when
# //  ModelSim SE 6.1 Beta Feb 26 2005 Linux 2.6.9-5.0.3.ELhugemem
# //
# //  Copyright Mentor Graphics Corporation 2005
# //              All Rights Reserved.
# //
# //  THIS WORK CONTAINS TRADE SECRET AND
# //  PROPRIETARY INFORMATION WHICH IS THE PROPERTY
# //  OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# Loading work.when
# do when.do
# Need to Save : 0
# when Stopping to allow checkpoint at 290 # Simulation stop requested.
# Resume macro at 290
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 610
# Simulation stop requested.
# Resume macro at 610
# Out of run command. Do checkpoint here # Need to Save : 0
# when Stopping to allow checkpoint at 930
# Simulation stop requested.
# Resume macro at 930
# Out of run command. Do checkpoint here
# Done at time                 1000
# ** Note: $finish    : when.v(14)
#    Time: 1 us  Iteration: 0  Instance: /when
```

Cell Libraries

Model Technology passed the ASIC Council’s Verilog test suite and achieved the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors’ Verilog cell libraries are compatible with ModelSim Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 for details on timing constraints. ModelSim Verilog fully implements specify blocks and timing constraints as defined in IEEE Std 1364 along with some Verilog-XL compatible extensions.

SDF Timing Annotation

ModelSim Verilog supports timing annotation from Standard Delay Format (SDF) files. See Standard Delay Format (SDF) Timing Annotation for details.

Delay Modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
   input a, b;
   output y;
   and(y, a, b);
   specify
      (a => y) = 5;
      (b => y) = 5;
   endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364). This is the default behavior, but you can specify alternate delay modes with compiler directives and arguments. These arguments and directives are compatible with Verilog-XL. Compiler delay mode arguments take precedence over delay mode directives in the source code.

Distributed Delay Mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay_mode_distributed** compiler argument or the **`delay_mode_distributed** compiler directive.

Path Delay Mode

In path delay mode the distributed delays are set to zero in any module that contains a path delay. Select this delay mode with the **+delay_mode_path** compiler argument or the **`delay_mode_path** compiler directive.

Unit Delay Mode

In unit delay mode the non-zero distributed delays are set to one unit of simulation resolution (determined by the minimum time_precision argument in all ‘timescale directives in your design or the value specified with the -t argument to vsim), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler argument or the **`delay_mode_unit** compiler directive.

Zero Delay Mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_zero** compiler argument or the **`delay_mode_zero** compiler directive.

System Tasks and Functions

ModelSim supports system tasks and functions as follows:

- All system tasks and functions defined in IEEE Std 1364
- Some system tasks and functions defined in SystemVerilog IEEE std p1800-2005 LRM
- Several system tasks and functions that are specific to ModelSim

- Several non-standard, Verilog-XL system tasks

The system tasks and functions listed in this section are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI), Verilog Procedural Interface (VPI), or the SystemVerilog DPI (Direct Programming Interface). If the simulator issues warnings regarding undefined system tasks or functions, then it is likely that these tasks or functions are defined by a PLI/VPI application that must be loaded by the simulator.

IEEE Std 1364 System Tasks and Functions

The following system tasks and functions are described in detail in the IEEE Std 1364.

Timescale tasks	**Simulator control tasks**	**Simulation time functions**	**Command line input**
$printtimescale	$finish	$realtime	$test$plusargs
$timeformat	$stop	$stime	$value$plusargs
		$time	

Probabilistic distribution functions	**Conversion functions**	**Stochastic analysis tasks**	**Timing check tasks**
$dist_chi_square	$bitstoreal	$q_add	$hold
$dist_erlang	$itor	$q_exam	$nochange
$dist_exponential	$realtobits	$q_full	$period
$dist_normal	$rtoi	$q_initialize	$recovery
$dist_poisson	$signed	$q_remove	$setup
$dist_t	$unsigned		$setuphold
$dist_uniform			$skew
$random			$width[1]
			$removal
			$recrem

[1] Verilog-XL ignores the threshold argument even though it is part of the Verilog spec. ModelSim does not ignore this argument. Be careful that you don’t set the threshold argument greater-than-or-equal to the limit argument as that essentially disables the $width check. Note too that you cannot override the threshold argument via SDF annotation.

Display tasks

$display
$displayb
$displayh
$displayo
$monitor
$monitorb
$monitorh
$monitoro
$monitoroff
$monitoron
$strobe
$strobeb
$strobeh
$strobeo
$write
$writeb
$writeh
$writeo

PLA modeling tasks

$async$and$array
$async$nand$array
$async$or$array
$async$nor$array
$async$and$plane
$async$nand$plane
$async$or$plane
$async$nor$plane
$sync$and$array
$sync$nand$array
$sync$or$array
$sync$nor$array
$sync$and$plane
$sync$nand$plane
$sync$or$plane
$sync$nor$plane

Value change dump (VCD) file tasks

$dumpall
$dumpfile
$dumpflush
$dumplimit
$dumpoff
$dumpon
$dumpvars
$dumpportson
$dumpportsoff
$dumpportsall
$dumpportsflush
$dumpports
$dumpportslimit

File I/O tasks

$fclose
$fdisplay
$fdisplayb
$fdisplayh
$fdisplayo
$feof
$ferror
$fflush
$fgetc
$fgets
$fmonitor
$fmonitorb
$fmonitorh
$fmonitoro
$fopen
$fread
$fscanf
$fseek
$fstrobe
$fstrobeb
$fstrobeh
$fstrobeo
$ftell
$fwrite
$fwriteb
$fwriteh
$fwriteo
$readmemb
$readmemh
$rewind
$sdf_annotate
$sformat
$sscanf
$swrite
$swriteb
$swriteh
$swriteo
$ungetc

SystemVerilog System Tasks and Functions

The following ModelSim-supported system tasks and functions are described in detail in the SystemVerilog IEEE Std p1800-2005 LRM.

Expression size function	Range function
$bits	$isunbounded

Shortreal conversions	Array querying functions
$shortrealbits	$dimensions
$bitstoshortreal	$left
	$right
	$low

Shortreal conversions	**Array querying functions**
	$high
	$increment
	$size

Reading packed data functions	**Writing packed data functions**	**Other functions**
$readmemb	$writememb	$root
$readmemh	$writememh	$unit

System Tasks and Functions Specific to the Tool

The following system tasks and functions are specific to ModelSim. They are not included in the IEEE Std 1364 nor are they likely supported in other simulators. Their use may limit the portability of your code.

```
$coverage_save(<filename>, [<instancepath>], [<xml_output>])
```

The $coverage_save() system function saves Code Coverage information to a file during a batch run that typically would terminate via the $finish call. It also returns a “1” to indicate that the coverage information was saved successfully or a “0” to indicate an error (unable to open file, instance name not found, etc.)

If you don’t specify <instancepath>, ModelSim saves all coverage data in the current design to the specified file. If you do specify <instancepath>, ModelSim saves data on that instance, and all instances below it (recursively), to the specified file.

If set to 1, the [<xml_output>] argument specifies that the output be saved in XML format.

See Measuring Coverage for more information on Code Coverage.

```
$init_signal_driver
```

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net onto an existing VHDL signal or Verilog net. This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_driver for complete details.

```
$init_signal_spy
```

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net onto an existing Verilog register or VHDL signal. This system task allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench). See $init_signal_spy for complete details.

`$psprintf()`

The $psprintf() system function behaves like the $sformat() file I/O task except that the string result is passed back to the user as the function return value for $psprintf(), not placed in the first argument as for $sformat(). Thus $psprintf() can be used where a string is valid. Note that at this time, unlike other system tasks and functions, $psprintf() cannot be overridden by a user-defined system function in the PLI.

`$signal_force`

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register or net. This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench). A $signal_force works the same as the force command with the exception that you cannot issue a repeating force. See $signal_force for complete details.

`$signal_release`

The $signal_release() system task releases a value that had previously been forced onto an existing VHDL signal or Verilog register or net. A $signal_release works the same as the noforce command. See $signal_release for complete details.

`$sdf_done`

This task is a "cleanup" function that removes internal buffers, called MIPDs, that have a delay value of zero. These MIPDs are inserted in response to the **-v2k_int_delay** argument to the vsim command. In general the simulator will automatically remove all zero delay MIPDs. However, if you have $sdf_annotate() calls in your design that are not getting executed, the zero-delay MIPDs are not removed. Adding the $sdf_done task after your last $sdf_annotate() will remove any zero-delay MIPDs that have been created.

Verilog-XL Compatible System Tasks and Functions

ModelSim supports a number of Verilog-XL specific system tasks and functions.

Supported Tasks and Functions Mentioned in IEEE Std 1364

The following supported system tasks and functions, though not part of the IEEE standard, are described in an annex of the IEEE Std 1364.

$countdrivers
$getpattern
$sreadmemb
$sreadmemh

Supported Tasks not Described in the IEEE Std 1364

The following system tasks are also provided for compatibility with Verilog-XL, though they are not described in the IEEE Std 1364.

```
$deposit(variable, value);
```

This system task sets a Verilog register or net to the specified value. **variable** is the register or net to be changed; **value** is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim **force -deposit** command.

```
$disable_warnings("<keyword>"[,<module_instance>...]);
```

This system task instructs ModelSim to disable warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module instance, ModelSim disables warnings for the entire simulation.

```
$enable_warnings("<keyword>"[,<module_instance>...]);
```

This system task enables warnings about timing check violations or triregs that acquire a value of 'X' due to charge decay. <keyword> may be **decay** or **timing**. You can specify one or more module instance names. If you don't specify a module_instance, ModelSim enables warnings for the entire simulation.

```
$system("<operating system shell command>");
```

This system task executes the specified operating system shell command and displays the result. For example, to list the contents of the working directory on Unix:

$system("ls");

Supported Tasks that Have Been Extended

The following system tasks are extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as in Verilog-XL.

```
$recovery(reference event, data_event, removal_limit, recovery_limit,
[notifier], [tstamp_cond], [tcheck_cond], [delayed_reference],
[delayed_data])
```

The $recovery system task normally takes a recovery_limit as the third argument and an optional notifier as the fourth argument. By specifying a limit for both the third and fourth arguments, the $recovery timing check is transformed into a combination removal and recovery timing check similar to the $recrem timing check. The only difference is that the removal_limit and recovery_limit are swapped.

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier],
[tstamp_cond], [tcheck_cond], [delayed_clk], [delayed_data])
```

The tstamp_cond argument conditions the data_event for the setup check and the clk_event for the hold check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The tcheck_cond argument conditions the data_event for the hold check and the clk_event for the setup check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The delayed_clk argument is a net that is continuously assigned the value of the net specified in the clk_event. The delay is non-zero if the setup_limit is negative, zero otherwise.

The delayed_data argument is a net that is continuously assigned the value of the net specified in the data_event. The delay is non-zero if the hold_limit is negative, zero otherwise.

The delayed_clk and delayed_data arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the delayed_clk and delayed_data nets in place of the normal clk and data nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for delayed_clk and delayed_data such that the correct data is latched as long as a timing constraint has not been violated. See Negative Timing Check Limits for more details.

Unsupported Verilog-XL System Tasks

The following system tasks are Verilog-XL system tasks that are not implemented in ModelSim Verilog, but have equivalent simulator commands.

```
$input("filename")
```

This system task reads commands from the specified filename. The equivalent simulator command is **do <filename>**.

```
$list[(hierarchical_name)]
```

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the structure pane of the Workspace. The corresponding source code is displayed in a Source window.

```
$reset
```

This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

```
$restart("filename")
```

This system task sets the simulation to the state specified by filename, saved in a previous call to $save. The equivalent simulator command is **restore <filename>**.

```
$save("filename")
```

This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

```
$scope(hierarchical_name)
```

This system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is **environment <pathname>**.

```
$showscopes
```

This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

```
$showvars
```

This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

Compiler Directives

ModelSim Verilog supports all of the compiler directives defined in the IEEE Std 1364, some Verilog-XL compiler directives, and some that are proprietary. The SystemVerilog IEEE Std P1800-2005 version of the *‘define* and *‘include* compiler directives are not currently supported.

Many of the compiler directives (such as **`timescale**) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a **`resetall** directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line could be significant. For example, if you have a file that defines some common macros for the entire design, then you might need to place it first in the list of files to be compiled.

The **`resetall** directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364):

```
`celldefine
‘default_decay_time
`default_nettype
`delay_mode_distributed
`delay_mode_path
`delay_mode_unit
`delay_mode_zero
`protected
`timescale
`unconnected_drive
`uselib
```

ModelSim Verilog implicitly defines the following macro:

```
`define MODEL_TECH
```

IEEE Std 1364 Compiler Directives

The following compiler directives are described in detail in the IEEE Std 1364.

```
`celldefine
`default_nettype
`define
`else
`elsif
`endcelldefine
`endif
`ifdef
‘ifndef
`include
‘line
`nounconnected_drive
`resetall
`timescale
`unconnected_drive
`undef
```

Compiler Directives for vlog

The following directives are specific to ModelSim and are not compatible with other simulators (see note below).

```
‘protect ... ‘endprotect
```

This directive pair allows you to encrypt selected regions of your source code. The code in **`protect** regions has all debug information stripped out. This behaves exactly as if using the **-nodebug** argument except that it applies to selected regions of code rather than the whole file. This enables usage scenarios such as making module ports, parameters, and specify blocks publicly visible while keeping the implementation private.

The **`protect** directive is ignored by default unless you use the **+protect** argument to vlog. Once compiled, the original source file is copied to a new file in the current work directory. The name of the new file is the same as the original file with a "p" appended to the suffix. For example, "top.v" is copied to "top.vp". This new file can be delivered and used as a replacement for the original source file.

A usage scenario might be that a vendor will use the **`protect / `endprotect** directives on a module or a portion of a module in a file named *filename.v*. They will compile it with **vlog +protect filename.v** to produce a new file named *filename.vp*. You can compile *filename.vp* just like any other verilog file. The protection is not compatible between tools, so the vendor mush ship you a different *filename.vp* than they ship to some who uses a different simulator.

The **+protect** argument is not required when compiling *.vp* files because the **`protect** directives are converted to **`protected** directives which are processed even if **+protect** is omitted.

`protect** and **`protected directives cannot be nested.

If any **`include** directives occur within a protected region, the compiler generates a copy of the include file with a ".vp" suffix and protects the entire contents of the include file.

If errors are detected in a protected region, the error message always reports the first line of the protected block.

Though other simulators have a **`protect** directive, the algorithm ModelSim uses to encrypt source files is different. Hence, even though an uncompiled source file with **`protect** is compatible with another simulator, once the source is compiled in ModelSim, you could not simulate it elsewhere.

Verilog-XL Compatible Compiler Directives

The following compiler directives are provided for compatibility with Verilog-XL.

```
‘default_decay_time <time>
```

This directive specifies the default decay time to be used in trireg net declarations that do not explicitly declare a decay time. The decay time can be expressed as a real or integer number, or as "infinite" to specify that the charge never decays.

```
`delay_mode_distributed
```

This directive disables path delays in favor of distributed delays. See Delay Modes for details.

```
`delay_mode_path
```

This directive sets distributed delays to zero in favor of path delays. See Delay Modes for details.

```
`delay_mode_unit
```

This directive sets path delays to zero and non-zero distributed delays to one time unit. See Delay Modes for details.

```
`delay_mode_zero
```

This directive sets path delays and distributed delays to zero. See Delay Modes for details.

```
`uselib
```

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler arguments. See Verilog-XL uselib Compiler Directive for details.

The following Verilog-XL compiler directives are silently ignored by ModelSim Verilog. Many of these directives are irrelevant to ModelSim Verilog, but may appear in code being ported from Verilog-XL.

```
`accelerate
`autoexpand_vectornets
`disable_portfaults
`enable_portfaults
`expand_vectornets
`noaccelerate
`noexpand_vectornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in ModelSim Verilog. These are not implemented in ModelSim Verilog, and any code containing these directives may behave differently in ModelSim Verilog than in Verilog-XL.

```
`default_trireg_strength
`signed
`unsigned
```

Sparse Memory Modeling

Sparse memories are a mechanism for allocating storage for memory elements only when they are needed. You mark which memories should be treated as sparse, and ModelSim dynamically allocates memory for the accessed addresses during simulation.

Sparse memories are more efficient in terms of memory consumption, but access times to sparse memory elements during simulation are slower. Thus, sparse memory modeling should be used only on memories whose active addresses are "few and far between."

There are two methods of enabling sparse memories:

- “Manually” by inserting attributes or meta-comments in your code
- Automatically by setting the SparseMemThreshhold variable in the *modelsim.ini* file

Manually Marking Sparse Memories

You can mark memories in your code as sparse using either the *mti_sparse* attribute or the *sparse* meta-comment. For example:

```
(* mti_sparse *) reg mem [0:1023]; // Using attribute
reg /*sparse*/ [0:7] mem [0:1023]; // Using meta-comment
```

The meta-comment syntax is supported for compatibility with other simulators.

Automatically Enabling Sparse Memories

Using the SparseMemThreshhold .ini variable, you can instruct ModelSim to mark as sparse any memory that is a certain size. Consider this example:

If SparseMemThreshold = 2048 then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

Combining Automatic and Manual Modes

Because *mti_sparse* is a Verilog 2001 attribute that accepts values, you can enable automatic sparse memory modeling but still control individual memories within your code. Consider this example:

If SparseMemThreshold = 2048 then

```
reg mem[0:2047]; // will be marked as sparse automatically
reg mem[0:2046]; // will not be marked as sparse
```

However, you can override this automatic behavior using *mti_sparse* with a value:

```
(* mti_sparse = 0 *) reg mem[0:2047];
// will *not* be marked as sparse even though SparseMemThreshold = 2048

(* mti_sparse = 1*) reg mem[0:2046];
// will be marked as sparse even though SparseMemThreshold = 2048
```

Determining Which Memories Were Implemented as Sparse

To identify which memories were implemented as sparse, use this command:

write report -l

The write report command lists summary information about the design, including sparse memory handling. You would issue this command if you aren’t certain whether a memory was successfully implemented as sparse or not. For example, you might add a /*sparse*/ metacomment above a multi-D SystemVerilog memory, which we don't support. In that case, the simulation will function correctly, but ModelSim will use a non-sparse implementation of the memory.

If you are planning to optimize your design with vopt, be sure to use the **+acc** argument in order to make the sparse memory visible, thus allowing the **write report -l** command to report the sparse memory.

Limitations

There are certain limitations that exist with sparse memories:

- Sparse memories can have only one packed dimension. For example:

  ```
  reg [0:3] [2:3] mem [0:1023]
  ```

  has two packed dimensions and cannot be marked as sparse.

- Sparse memories can have only one unpacked dimension. For example:

  ```
  reg [0:1] mem [0:1][0:1023]
  ```

  has two unpacked dimensions and cannot be marked as sparse.

- Dynamic and associative arrays cannot be marked as sparse.
- Memories defined within a structure cannot be marked as sparse.
- PLI functions that get the pointer to the value of a memory will not work with sparse memories. For example, using the **tf_nodeinfo()** function to implement $fread or $fwrite will not work, because ModelSim returns a NULL pointer for tf_nodeinfo() in the case of sparse memories.
- Memories that have parameterized dimensions like the following example:

  ```
  parameter MYDEPTH = 2048;
  reg [31:0] mem [0:MYDEPTH-1];
  ```

  cannot be processed as a sparse memory *unless* the design has been optimized with the vopt command. In optimized designs, the memory will be implemented as a sparse memory, and all parameter overrides to that MYDEPTH parameter will be treated correctly.

Verilog PLI/VPI and SystemVerilog DPI

ModelSim supports the use of the Verilog PLI (Programming Language Interface) and VPI (Verilog Procedural Interface) and the SystemVerilog DPI (Direct Programming Interface). These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. For more information on the ModelSim implementation, see Verilog PLI/VPI/DPI.

Chapter 8
SystemC Simulation

This chapter describes how to compile and simulate SystemC designs with ModelSim. ModelSim implements the SystemC language based on the Open SystemC Initiative (OSCI) SystemC 2.1 reference simulator. It is recommended that you obtain the OSCI functional specification, or the latest version of the SystemC Language Reference Manual as a reference manual. Visit *http://www.systemc.org* for details.

Note

The functionality described in this chapter requires a systemc license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

In addition to the functionality described in the OSCI specification, ModelSim for SystemC includes the following features:

- Single common Graphic Interface for SystemC and HDL languages.
- Extensive support for mixing SystemC, VHDL, and Verilog in the same design (SDF annotation for HDL only). For detailed information on mixing SystemC with HDL see Mixed-Language Simulation.

Supported Platforms and Compiler Versions

SystemC runs on a subset of ModelSim supported platforms. The table below shows the currently supported platforms and compiler versions:

Table 8-1. Supported Platforms for SystemC

Platform	Supported compiler versions	32-bit support	64-bit support
HP-UX 11.0 or later	aCC 3.45 with associated patches	yes	no
RedHat Linux 7.2 and 7.3, RedHat Linux Enterprise version 2.1	gcc 3.2.3, gcc 4.0.2	yes	no
AMD64 / SUSE Linux Enterprise Server 9.0, 9.1 or Red Hat Enterprise Linux 3, 4	gcc 4.0.2 VCO is linux (32-bit binary) VCO is linux_x86_64 (64-bit binary)	yes	yes
Solaris 8, 9, and 10	gcc 3.3	yes	no

Table 8-1. Supported Platforms for SystemC

Platform	Supported compiler versions	32-bit support	64-bit support
Windows 2000 and XP	Minimalist GNU for Windows (MinGW) gcc 3.3.1	yes	no

Note

ModelSim SystemC has been tested with the gcc versions available from the install tree. Customized versions of gcc may cause problems. We strongly encourage you to use the supplied gcc versions.

Building gcc with Custom Configuration Options

We only test with our default options. If you use advanced gcc configuration options, we cannot guarantee that ModelSim will work with those options.

To use a custom gcc build, set the CppPath variable in the *modelsim.ini* file. This variable specifies the pathname to the compiler binary you intend to use.

When using a custom gcc, ModelSim requires that the custom gcc be built with several specific configuration options. These vary on a per-platform basis as shown in the following table:

Table 8-2.

Platform	Mandatory configuration options
Linux	none
Solaris	--with-gnu-ld --with-ld=/path/to/binutils-2.14/bin/ld --with-gnu-as --with-as=/path/to/binutils-2.14/bin/as
HP-UX	N/A
Win32 (MinGW)	--with-gnu-ld --with-gnu-as Do NOT build with: --enable-sjlj-exceptions option as it can cause problems with catching exceptions thrown from SC_THREAD and SC_CTHREAD *ld.exe* and *as.exe* should be installed into the *<install_dir>/bin* before building gcc. *ld* and *as* are available in the binutils package. ModelSim uses binutils 2.13.90-20021006-2.

If you don't have a GNU binutils2.14 assembler and linker handy, you can use the as and ld programs distributed with ModelSim. They are located inside the built-in gcc in directory *<install_dir>/modeltech/gcc-3.2-<mtiplatform>/lib/gcc-lib/<gnuplatform>/3.2*.

By default ModelSim also uses the following options when configuring built-in gcc.

- --disable-nls
- --enable-languages=c,c++

These are not mandatory, but they do reduce the size of the gcc installation.

HP Limitations for SystemC

HP is supported for SystemC with the following limitations:

- variables are not supported
- aggregates are not supported
- objects must be explicitly named, using the same name as their object, in order to debug

SystemC simulation objects such as modules, primitive channels, and ports can be explicitly named by passing a name to the constructors of said objects. If an object is not constructed with an explicit name, then the OSCI reference simulator generates an internal name for it, using names such as "signal_0", "signal_1", etc.

Required Patch for HP-UX 11.11

If you are running on HP-UX 11.11, you must have the following patch installed:

B.11.11.0306 Gold Base Patches for HP-UX 11i, June 2003.

Usage Flow for SystemC-Only Designs

ModelSim allows users to simulate SystemC, either alone or in combination with other VHDL/Verilog modules. The following is an overview of the usage flow for strictly SystemC designs. More detailed instructions are presented in the sections that follow.

1. Create and map the working design library with the **vlib** and **vmap** statements, as appropriate to your needs.
2. Modify the SystemC source code, including the following highlights:
   a. Replace **sc_main()** with an SC_MODULE, and potentially add a process to contain any testbench code

   b. Replace **sc_start()** by using the run command in the GUI

   c. Remove calls to **sc_initialize()**

   d. Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro

   See Modifying SystemC Source Code for a complete list of all modifications.

3. Analyze the SystemC source using sccom. **sccom** invokes the native C++ compiler to create the C++ object files in the design library.

   See Using sccom in Addition to the Raw C++ Compiler for information on when you are required to use **sccom** vs. another C++ compiler.

4. Perform a final link of the C++ source using sccom -link. This process creates a shared object file in the current work library which will be loaded by **vsim** at runtime.

   sccom -link must be re-run before simulation if any new **sccom** compiles were performed.

5. Simulate the design using the standard **vsim command.**

6. Simulate the design using the **run** command, entered at the **vsim** command prompt.

7. Debug the design using ModelSim GUI features, including the Source and Wave windows.

Compiling SystemC Files

To compile SystemC designs, you must

- create a design library
- modify the SystemC source code
- run the sccom SystemC compiler
- run the sccom SystemC linker (sccom -link)

Creating a Design Library for SystemC

Before you can compile your design, you must create a library in which to store the compilation results. Use vlib to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

The **work** library is actually a subdirectory named work. This subdirectory contains a special file named _info. Do not create libraries using UNIX commands – always use the vlib command.

See Design Libraries for additional information on working with libraries.

Modifying SystemC Source Code

Several modifications must be applied to your original SystemC source code. To see example code containing the modifications listed below, see Code Modification Examples.

Converting sc_main() to a Module

In order for ModelSim to run the SystemC/C++ source code, the control function of **sc_main()** must be replaced by a constructor, SC_CTOR(), placed within a module at the top level of the design (see **mytop** in Example 8-1). In addition:

- any testbench code inside **sc_main()** should be moved to a process, normally an SC_THREAD process.
- all C++ variables in **sc_main()**, including SystemC primitive channels, ports, and modules, must be defined as members of sc_module. Therefore, initialization must take place in the SC_CTOR. For example, all **sc_clock()** and **sc_signal()** initializations must be moved into the constructor.

Replacing the sc_start() Function with the Run Command and Options

ModelSim uses the **run** command and its options in place of the **sc_start()** function. If **sc_main()** has multiple **sc_start()** calls mixed in with the testbench code, then use an **SC_THREAD()** with wait statements to emulate the same behavior. An example of this is shown below.

Removing Calls to sc_initialize()

vsim calls **sc_initialize()** by default at the end of elaboration, so calls to **sc_initialize()** are unnecessary.

Exporting All Top Level SystemC Modules

For SystemC designs, you must export all top level modules in your design to ModelSim. You do this with the **SC_MODULE_EXPORT(<sc_module_name>)** macro. SystemC templates are not supported as top level or boundary modules. See Templatized SystemC Modules. The **sc_module_name** is the name of the top level module to be simulated in ModelSim. You must specify this macro in a C++ source (*.cpp*) file. If the macro is contained in a header file instead of a C++ source file, an error may result.

Explicitly Naming Signals, Ports, And Modules for HP-UX

Important: Verify that SystemC signals, ports, and modules are explicitly named to avoid port binding and debugging errors.

Code Modification Examples

Example 8-1. Converting sc_main to a Module

The following is a simple example of how to convert **sc_main** to a module and elaborate it with **vsim**.

Table 8-3. Simple conversion - sc_main to Module

<table>
<tr><th>Original OSCI code #1 (partial)</th><th>Modified code #1 (partial)</th></tr>
<tr><td><pre>
int sc_main(int argc, char*
argv[])
{
     sc_signal<bool> mysig;
     mymod mod("mod");
     mod.outp(mysig);

     sc_start(100, SC_NS);
}
</pre></td><td><pre>
SC_MODULE(mytop)
{
     sc_signal<bool> mysig;
     mymod mod;

     SC_CTOR(mytop)
         : mysig("mysig"),
           mod("mod")
     {
         mod.outp(mysig);
     }
};

SC_MODULE_EXPORT(mytop);
</pre></td></tr>
</table>

The run command equivalent to the sc_start(100, SC_NS) statement is:

```
run 100 ns
```

Example 8-2. Using sc_main and Signal Assignments

This next example is slightly more complex, illustrating the use of **sc_main()** and signal assignments, and how you would get the same behavior using ModelSim.

Table 8-4. Using sc_main and Signal Assignments

OSCI code #2 (partial)	Modified code #2 (partial)

int sc_main(int, char**)
{
   sc_signal<bool> reset;
   counter_top top("top");
   sc_clock CLK("CLK", 10, SC_NS,
                  0.5, 0.0, SC_NS,
   false);
 	
   top.reset(reset);
 	
   reset.write(1);
   sc_start(5, SC_NS);
   reset.write(0);
   sc_start(100, SC_NS);
   reset.write(1);
   sc_start(5, SC_NS);
   reset.write(0);
   sc_start(100, SC_NS);
}
	SC_MODULE(new_top)
	{
	   sc_signal<bool> reset;
	   counter_top top;
	   sc_clock CLK;
	
	   void sc_main_body();
	
	   SC_CTOR(new_top)
	     : reset("reset"),
	     top("top")
	     CLK("CLK", 10, SC_NS, 0.5, 0.0, SC_NS, false)
	     {
	     top.reset(reset);
	     SC_THREAD(sc_main_body);
	     }
	};
	
	void
	new_top::sc_main_body()
	{
	   reset.write(1);
	   wait(5, SC_NS);
	   reset.write(0);
	   wait(100, SC_NS);
	   reset.write(1);
	   wait(5, SC_NS);
	   reset.write(0);
	   wait(100, SC_NS);
	}
	
	SC_MODULE_EXPORT(new_top);

Example 8-3. Using an SCV Transaction Database

One last example illustrates the correct way to modify a design using an SCV transaction database. ModelSim requires that the transaction database be created before calling the constructors on the design subelements. The example is as follows:

Table 8-5.

<table>
<tr><th>Original OSCI code # 3 (partial)</th><th>Modified ModelSim code #3 (partial)</th></tr>
<tr><td><pre>
int sc_main(int argc, char*
argv[])
{
   scv_startup();
   scv_tr_text_init();
   scv_tr_db db("my_db");
   scv_tr_db
   db::set_default_db(&db);

   sc_clock clk
   ("clk",20,0.5,0,true);
   sc_signal<bool> rw;
   test t("t");

   t.clk(clk);;
   t.rw(rw);

   sc_start(100);
}
</pre></td><td><pre>
SC_MODULE(top)
{
   sc_signal<bool>* rw;
   test* t;

   SC_CTOR(top)
   {
     scv_startup();
     scv_tr_text_init()
     scv_tr_db* db = new
     scv_tr_db("my_db");
     scv_tr_db::set_default_db(db):;

     clk = new
     sc_clock("clk",20,0.5,0,true);
     rw = new sc_signal<bool> ("rw");
     t = new test("t");
   }
};

SC_MODULE_EXPORT(new_top);
</pre></td></tr>
</table>

Take care to preserve the order of functions called in **sc_main()** of the original code.

Sub-elements cannot be placed in the initializer list, since the constructor body must be executed prior to their construction. Therefore, the sub-elements must be made pointer types, created with "new" in the SC_CTOR() module.

Invoking the SystemC Compiler

ModelSim compiles one or more SystemC design units with a single invocation of sccom, the SystemC compiler. The design units are compiled in the order that they appear on the command line. For SystemC designs, all design units must be compiled just as they would be for any C++ compilation. An example of an **sccom** command might be:

```
sccom -I ../myincludes mytop.cpp mydut.cpp
```

Compiling Optimized and/or Debug Code

By default, **sccom** invokes the C++ compiler (g++ or aCC) without any optimizations. If desired, you can enter any g++/aCC optimization arguments at the **sccom** command line.

Also, source level debug of SystemC code is not available by default in ModelSim. To compile your SystemC code for debug, use the g++/aCC **-g** argument on the **sccom** command line.

Specifying an Alternate g++ Installation

We recommend using the version of g++ that is shipped with ModelSim on its various supported platforms. However, if you want to use your own installation, you can do so by setting the CppPath variable in the *modelsim.ini* file to the g++ executable location.

For example, if your g++ executable is installed in */u/abc/gcc-3.2/bin*, then you would set the variable as follows:

```
CppPath /u/abc/gcc-3.2/bin/g++
```

Maintaining Portability Between OSCI and the Simulator

If you intend to simulate on both ModelSim and the OSCI reference simulator, you can use the MTI_SYSTEMC macro to execute the ModelSim specific code in your design only when running ModelSim. **Sccom** defines this macro by default during compile time.

Using the original and modified code shown in the example shown in Example 8-1, you might write the code as follows:

```
#ifdef MTI_SYSTEMC //If using the ModelSim simulator, sccom compiles this
SC_MODULE(mytop)
{
     sc_signal<bool> mysig;
     mymod mod;

     SC_CTOR(mytop)
         : mysig("mysig"),
           mod("mod")
     {
         mod.outp(mysig);
     }
};

SC_MODULE_EXPORT(top);

#else //Otherwise, it compiles this
int sc_main(int argc, char* argv[])
{
     sc_signal<bool> mysig;
     mymod mod("mod");
     mod.outp(mysig);

     sc_start(100, SC_NS);
}
#endif
```

Restrictions on Compiling with HP aCC

ModelSim uses the **aCC -AA** option by default when compiling C++ files on HP-UX. It does this so *cout* will function correctly in the *systemc.so* file. The **-AA** option tells **aCC** to use ANSI-compliant <iostream> rather than cfront-style <iostream.h>. Thus, all C++-based objects in a program must be compiled with **-AA**. This means you must use <iostream> and "using" clauses in your code. Also, you cannot use the **-AP** option, which is incompatible with **-AA**.

Switching Platforms and Compilation

Compiled SystemC libraries are platform dependent. If you move between platforms, you must remove all SystemC files from the working library and then recompile your SystemC source files. To remove SystemC files from the working directory, use the vdel command with the **-allsystemc** argument.

If you attempt to load a design that was compiled on a different platform, an error such as the following occurs:

```
# vsim work.test_ringbuf
# Loading work/systemc.so
# ** Error: (vsim-3197) Load of "work/systemc.so" failed:
work/systemc.so: ELF file data encoding not little-endian.
# ** Error: (vsim-3676) Could not load shared library
work/systemc.so for SystemC module 'test_ringbuf'.
# Error loading design
```

You can type **verror 3197** at the **vsim** command prompt and get details about what caused the error and how to fix it.

Using sccom in Addition to the Raw C++ Compiler

When compiling complex C/C++ testbench environments, it is common to compile code with many separate runs of the compiler. Often users compile code into archives (.a files), and then link the archives at the last minute using the -L and -l link options.

When using ModelSim's SystemC, you may wish to compile a portion of your C design using raw g++ or aCC instead of **sccom**. Perhaps you have some legacy code or some non-SystemC utility code that you want to avoid compiling with **sccom**. You can do this, however, some caveats and rules apply.

Rules for sccom Use

The rules governing when and how you must use **sccom** are as follows:

1. You must compile all code that references SystemC types or objects using sccom.

2. When using **sccom**, you should not use the -I compiler option to point the compiler at any search directories containing OSCI or any other vendor supplied SystemC header files. **sccom** does this for you accurately and automatically.
3. If you do use the raw C++ compiler to compile C/C++ functionality into archives or shared objects, you must then link your design using the -L and -l options with the **sccom -link** command. These options effectively pull the non-SystemC C/C++ code into a simulation image that is used at runtime.

Failure to follow the above rules can result in link-time or elaboration-time errors due to mismatches between the OSCI or any other vendor supplied SystemC header files and the ModelSim SystemC header files.

Rules for Using Raw g++ to Compile Non-SystemC C/C++ Code

If you use raw g++ to compile your non-systemC C/C++ code, the following rules apply:

1. The -fPIC option to g++ should be used during compilation with **sccom**.
2. For C++ code, you must use the built-in g++ delivered with ModelSim, or (if using a custom g++) use the one you built and specified with the CppPath .ini variable.

Otherwise binary incompatibilities may arise between code compiled by **sccom** and code compiled by raw g++.

Rules for Using Raw HP aCC to Compile Non-SystemC C/C++ Code

If you use HP's aCC compiler to compile your non-systemC C/C++ code, the following rules apply:

1. For C++ code, you should use the +Z and -AA options during compilation
2. You must use HP aCC version 3.45 or higher.

Issues with C++ Templates

Templatized SystemC Modules

Templatized SystemC modules are not supported for use at:

- the top level of the design
- the boundary between SystemC and higher level HDL modules (i.e. the top level of the SystemC branch)

To convert a top level templatized SystemC module, you can either specialize the module to remove the template, or you can create a wrapper module that you can use as the top module.

For example, let's say you have a templatized SystemC module as shown below:

```
template <class T>
class top : public sc_module
{
   sc_signal<T> sig1;
   ...
};
```

You can specialize the module by setting T = int, thereby removing the template, as follows:

```
class top : public sc_module
{
   sc_signal<int> sig 1;
   ...
};
```

Or, alternatively, you could write a wrapper to be used over the template module:

```
class modelsim_top : public sc_module
{
   top<int> actual_top;
   ...
};

SC_MODULE_EXPORT(modelsim_top);
```

Organizing Templatized Code

Suppose you have a class template, and it contains a certain number of member functions. All those member functions must be visible to the compiler when it compiles any instance of the class. For class templates, the C++ compiler generates code for each unique instance of the class template. Unless it can see the full implementation of the class template, it cannot generate code for it thus leaving the invisible parts as undefined. Since it is legal to have undefined symbols in a .so, **sccom -link** will not produce any errors or warnings. To make functions visible to the compiler, you must move them to the .h file.

Linking the Compiled Source

Once the design has been compiled, it must be linked using the sccom command with the **-link** argument.

The **sccom -link** command collects the object files created in the different design libraries, and uses them to build a shared library (.so) in the current work library or the library specified by the **-work** option. If you have changed your SystemC source code and recompiled it using **sccom**, then you must relink the design by running **sccom -link** before invoking **vsim**. Otherwise, your changes to the code are not recognized by the simulator. Remember that any dependent .a or .o files should be listed on the **sccom -link** command line before the .a or .o on which it depends. For more details on dependencies and other syntax issues, see sccom.

Simulating SystemC Designs

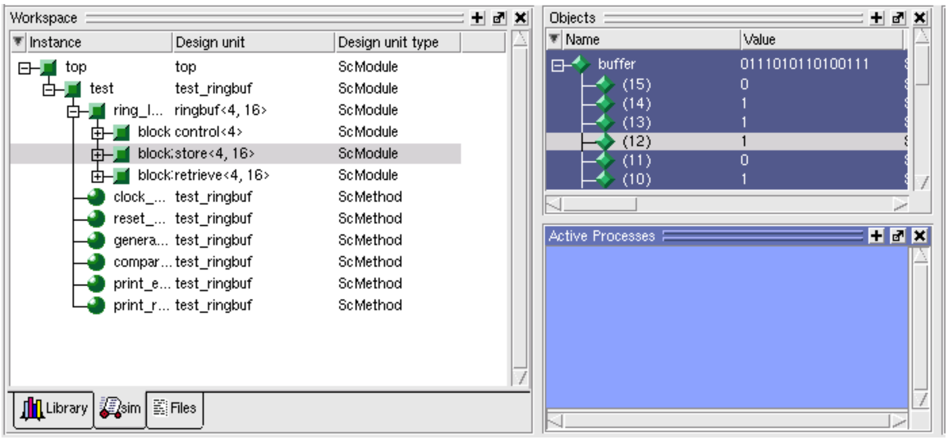
After compiling the SystemC source code, you can simulate your design with vsim.

Loading the Design

For SystemC, invoke vsim with the top-level module of the design. This example invokes vsim on a design named top:

```
vsim top
```

When the GUI comes up, you can expand the hierarchy of the design to view the SystemC modules. SystemC objects are denoted by green icons (see Design Object Icons and Their Meaning for more information).



Instance	Design unit	Design unit type
top	top	ScModule
test	test_ringbuf	ScModule
ring_l...	ringbuf<4, 16>	ScModule
block	control<4>	ScModule
block:	store<4, 16>	ScModule
block:	retrieve<4, 16>	ScModule
clock_...	test_ringbuf	ScMethod
reset_...	test_ringbuf	ScMethod
genera...	test_ringbuf	ScMethod
compar...	test_ringbuf	ScMethod
print_e...	test_ringbuf	ScMethod
print_r...	test_ringbuf	ScMethod

Name	Value
buffer	0111010110100111
(15)	0
(14)	1
(13)	1
(12)	1
(11)	0
(10)	1

To simulate from a command shell, without the GUI, invoke vsim with the -c option:

```
vsim -c <top_level_module>
```

Running Simulation

Run the simulation using the run command or select one of the **Simulate** > **Run** options from the menu bar.

SystemC Time Unit and Simulator Resolution

This section applies to SystemC only simulations. For simulations of mixed-language designs, the rules for how ModelSim interprets the resolution vary. See Simulator Resolution Limit for details on mixed-language simulations.

Two related yet distinct concepts are involved with determining the simulation resolution: the SystemC time unit and the simulator resolution. The following table describes the concepts, lists the default values, and defines the methods for setting/overriding the values.

Table 8-6.

	Description	Set by default as *.ini* file	Default value	Override default by
SystemC time unit	The unit of time used in your SystemC source code. You need to set this in cases where your SystemC default time unit is at odds with any other, non-SystemC segments of your design.	ScTimeUnit	1ns	ScTimeUnit *.ini* file variable or **sc_set_default_time_unit()** function before an sc_clock or sc_time statement.
Simulator resolution	The smallest unit of time measured by the simulator. If your delays get truncated, set the resolution smaller; this value must be less than or equal to the UserTimeUnit	Resolution	1ns	**-t** argument to vsim (This overrides all other resolution settings.) or **sc_set_time_resolution()** function or GUI: **Simulate** > **Start Simulation** > **Resolution**

Available settings for both time unit and resolution are: 1x, 10x, or 100x of fs, ps, ns, us, ms, or sec.

You can view the current simulator resolution by invoking the report command with the **simulator state** option.

Choosing Your Simulator Resolution

You should choose the coarsest simulator resolution limit possible that does not result in undesired rounding of your delays. However, the time precision should also not be set unnecessarily small, because in some cases performance will be degraded.

When deciding what to set the simulator's resolution to, you must keep in mind the relationship between the simulator's resolution and the SystemC time units specified in the source code. For example, with a time unit usage of:

```
sc_wait(10, SC_PS);
```

a simulator resolution of 10ps would be fine. No rounding off of the ones digits in the time units would occur. However, a specification of:

```
sc_wait(9, SC_PS);
```

would require you to set the resolution limit to 1ps in order to avoid inaccuracies caused by rounding.

Initialization and Cleanup of SystemC State-Based Code

State-based code should not be used in Constructors and Destructors. Constructors and Destructors should be reserved for creating and destroying SystemC design objects, such as sc_modules or sc_signals. State-based code should also not be used in the elaboration phase callbacks **before_end_of_elaboration()** and **end_of_elaboration()**.

The following virtual functions should be used to initialize and clean up state-based code, such as logfiles or the VCD trace functionality of SystemC. They are virtual methods of the following classes: sc_port_base, sc_module, sc_channel, and sc_prim_channel. You can think of them as phase callback routines in the SystemC language:

- before_end_of_elaboration () — Called after all constructors are called, but before port binding.
- end_of_elaboration () — Called at the end of elaboration after port binding. This function is available in the SystemC 2.1 reference simulator.
- start_of_simulation () — Called before simulation starts. Simulation-specific initialization code can be placed in this function.
- end_of_simulation () — Called before ending the current simulation session.

The call sequence for these functions with respect to the SystemC object construction and destruction is as follows:

1. Constructors
2. before_end_of_elaboration ()
3. end_of_elaboration ()
4. start_of_simulation ()
5. end_of_simulation ()
6. Destructors

Usage of Callbacks

The **start_of_simulation()** callback is used to initialize any state-based code. The corresponding cleanup code should be placed in the **end_of_simulation()** callback. These callbacks are only called during simulation by **vsim** and thus, are safe.

If you have a design in which some state-based code must be placed in the constructor, destructor, or the elaboration callbacks, you can use the **mti_IsVoptMode()** function to determine if the elaboration is being run by vopt. You can use this function to prevent **vopt** from executing any state-based code.

Debugging the Design

You can debug SystemC designs using all of ModelSim's debugging features, with the exception of the Dataflow window.

Viewable SystemC Types

Types (<type>) of the objects which may be viewed for debugging are the following:

Types

bool, sc_bit
sc_logic
sc_bv<width>
sc_lv<width>
sc_int<width>
sc_uint<width>
sc_fix
sc_fix_fast
sc_fixed<W,I,Q,O,N>
sc_fixed_fast<W,I,Q,O,N>
sc_ufix
sc_ufix_fast
sc_ufixed
sc_ufixed_fast
sc_signed
sc_unsigned
char, unsigned char
int, unsigned int
short, unsigned short
long, unsigned long
sc_bigint<width>
sc_biguint<width>
sc_ufixed<W,I,Q,O,N>
short, unsigned short
long long, unsigned long long
float
double
enum
pointer
array
class
struct
union

Types Used as Elements of C/C++ Composite Type

SystemC types used as elements in C/C++ composite types - such as array, struct, class, and union - are not debuggable.

Viewable SystemC Objects

Objects which may be viewed in SystemC for debugging purposes are as shown in the following table.

Table 8-7. Viewable SystemC Objects

Channels	Ports	Variables	Aggregates
sc_clock (a hierarchical channel) sc_event sc_export sc_mutex sc_fifo<type> sc_signal<type> sc_signal_rv<width> sc_signal_resolved tlm_fifo<type> User defined channels derived from sc_prim_channel	sc_in<type> sc_out<type> sc_inout<type> sc_in_rv<width> sc_out_rv<width> sc_inout_rv<width> sc_in_resolved sc_out_resolved sc_inout_resolved sc_in_clk sc_out_clk sc_inout_clk sc_fifo_in sc_fifo_out User defined ports derived from sc_port<> which is : • connected to a built-in channel • connected to a user-defined channel derived from an sc_prim_channel[1]	Module member variables of all C++ and SystemC built-in types (listed in the Types list below) are supported.	Aggregates of SystemC signals or ports. Only three types of aggregates are supported for debug: struct class array

[1] You must use a special macro to make these ports viewable for debugging. For details See MTI_SC_PORT_ENABLE_DEBUG.

MTI_SC_PORT_ENABLE_DEBUG

A user-defined port which is not connected to a built-in primitive channel is not viewable for debugging by default. You can make the port viewable if the actual channel connected to the port is a channel derived from an sc_prim_channel. If it is, you can add the macro MTI_SC_PORT_ENABLE_DEBUG to the channel class' public declaration area, as shown in this example:

```
class my_channel: public sc_prim_channel

{
...
public:
     MTI_SC_PORT_ENABLE_DEBUG

};
```

Waveform Compare with SystemC

Waveform compare supports the viewing of SystemC signals and variables. You can compare SystemC objects to SystemC, Verilog or VHDL objects.

For pure SystemC compares, you can compare any two signals that match type and size exactly; for C/C++ types and some SystemC types, sign is ignored for compares. Thus, you can compare char to unsigned char or sc_signed to sc_unsigned. All SystemC fixed-point types may be mixed as long as the total number of bits and the number of integer bits match.

Mixed-language compares are supported as listed in the following table:

Table 8-8. Mixed-language Compares

C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned
Verilog types	net, reg
VHDL types	bit, bit_vector, boolean, std_logic, std_logic_vector

The number of elements must match for vectors; specific indexes are ignored.

Source-Level Debug

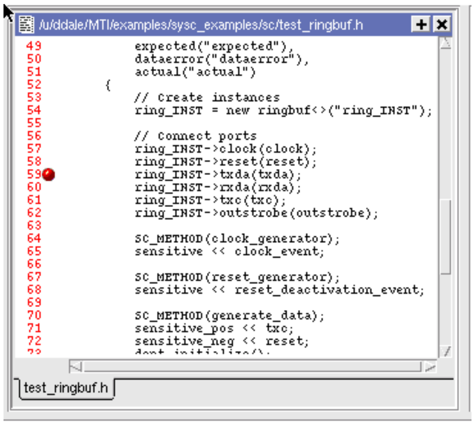
In order to debug your SystemC source code, you must compile the design for debug using the **-g** C++ compiler option. You can add this option directly to the sccom command line on a per run basis, with a command such as:

```
sccom mytop -g
```

Or, if you plan to use it every time you run the compiler, you can specify it in the *modelsim.ini* file with the **SccomCppOptions** variable. See SystemC Compiler Control Variables for more information.

The source code debugger, C Debug, is automatically invoked when the design is compiled for debug in this way.

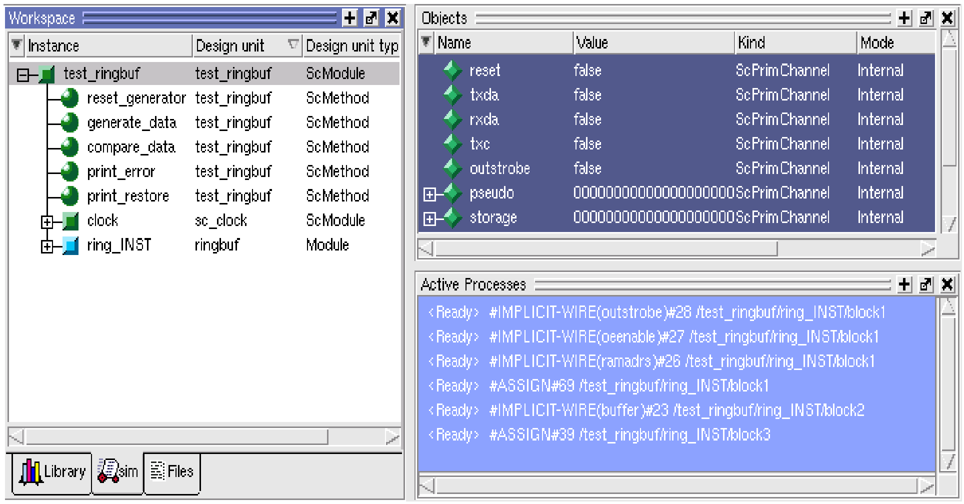
You can set breakpoints in a Source window, and single-step through your SystemC/C++ source code. .



```
            expected("expected"),
            dataerror("dataerror"),
            actual("actual")
        {
            // Create instances
            ring_INST = new ringbuf<>("ring_INST");

            // Connect ports
            ring_INST->clock(clock);
            ring_INST->reset(reset);
            ring_INST->txda(txda);
            ring_INST->rxda(rxda);
            ring_INST->txc(txc);
            ring_INST->outstrobe(outstrobe);

            SC_METHOD(clock_generator);
            sensitive << clock_event;

            SC_METHOD(reset_generator);
            sensitive << reset_deactivation_event;

            SC_METHOD(generate_data);
            sensitive_pos << txc;
            sensitive_neg << reset;
```

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Try to avoid setting breakpoints in constructors of SystemC objects; it may crash the debugger.

You can view and expand SystemC objects in the Objects pane and processes in the Active Processes pane.



Instance	Design unit	Design unit typ
test_ringbuf	test_ringbuf	ScModule
reset_generator	test_ringbuf	ScMethod
generate_data	test_ringbuf	ScMethod
compare_data	test_ringbuf	ScMethod
print_error	test_ringbuf	ScMethod
print_restore	test_ringbuf	ScMethod
clock	sc_clock	ScModule
ring_INST	ringbuf	Module

Name	Value	Kind	Mode
reset	false	ScPrimChannel	Internal
txda	false	ScPrimChannel	Internal
rxda	false	ScPrimChannel	Internal
txc	false	ScPrimChannel	Internal
outstrobe	false	ScPrimChannel	Internal
pseudo	00000000000000000000	ScPrimChannel	Internal
storage	00000000000000000000	ScPrimChannel	Internal

```
<Ready> #IMPLICIT-WIRE(outstrobe)#28 /test_ringbuf/ring_INST/block1
<Ready> #IMPLICIT-WIRE(oeenable)#27 /test_ringbuf/ring_INST/block1
<Ready> #IMPLICIT-WIRE(ramadrs)#26 /test_ringbuf/ring_INST/block1
<Ready> #ASSIGN#69 /test_ringbuf/ring_INST/block1
<Ready> #IMPLICIT-WIRE(buffer)#23 /test_ringbuf/ring_INST/block2
<Ready> #ASSIGN#39 /test_ringbuf/ring_INST/block3
```

SystemC Object and Type Display

This section contains information on how ModelSim displays certain objects and types, as they may differ from other simulators.

Support for Globals and Statics

Globals and statics are supported for ModelSim debugging purposes, however some additional naming conventions must be followed to make them viewable.

Naming Requirement

In order to make a global viewable for debugging purposes, the name given must match the declared signal name. An example:

```
sc_signal<bool> clock("clock");
```

For statics to be viewable, the name given must be fully qualified, with the module name and declared name, as follows:

```
<module_name>::<declared_name>
```

For example, the static data member "count" is viewable in the following code excerpt:

```
SC_MODULE(top)
{
   static sc_signal<floag> count; //static data member
   ....
}
sc_signal<float> top::count("top::count"); //static named in quotes
```

Viewing Global and Static Signals

ModelSim translates C++ scopes into a hierarchical arrangement. Since globals and statics exist at a level above ModelSim's scope, ModelSim must add a top level, **sc_root**, to all global and static signals. Thus, to view these static or global signals in ModelSim, you need to add **sc_root** to the hierarchical name for the signal. In the case of the above examples, the debugging statements for examining "top/count" (a static) and "clock" (a global) would be:

```
VSIM> examine /sc_root/top/count
VSIM> examine /sc_root/clock
```

Support for Aggregates

ModelSim supports aggregates of SystemC signals or ports. Three types of aggregates are supported: structures, classes, and arrays. Unions are not supported for debug. An aggregate of signals or ports will be shown as a signal of aggregate type. For example, an aggregate such as:

```
sc_signal <sc_logic> a[3];
```

is equivalent to:

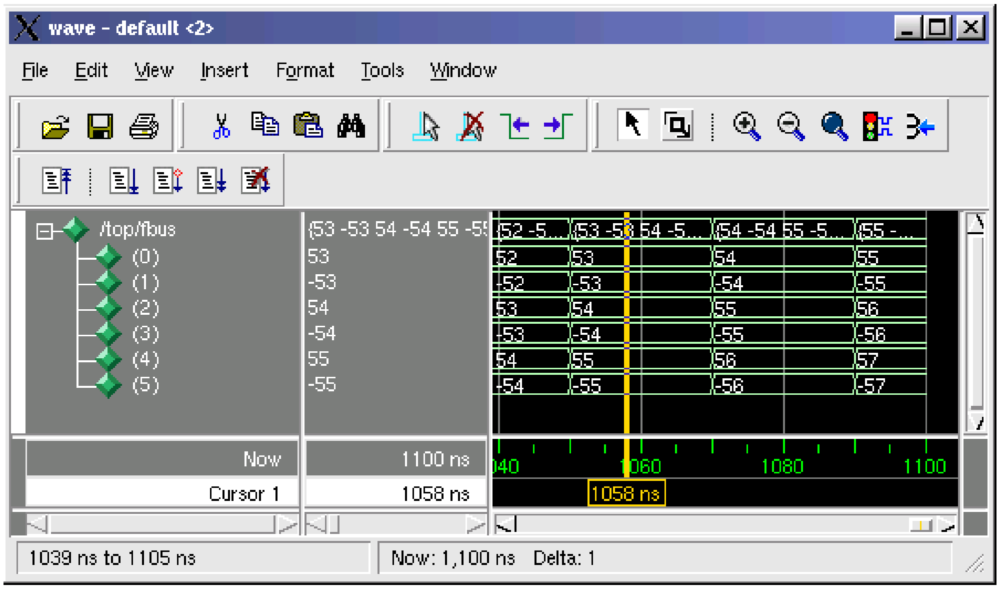
```
sc_signal <sc_lv<3>> a;
```

for debug purposes. ModelSim shows one signal - object "a" - in both cases.

The following aggregate

```
sc_signal <float> fbus [6];
```

when viewed in the Wave window, would appear as follows:



Viewing FIFOs

In ModelSim, the values contained in an sc_fifo appear in a definite order. The top-most or left-most value is always the next to be read from the FIFO. Elements of the FIFO that are not in use are not displayed.

Example of a signal where the FIFO has five elements:

```
# examine f_char
# {}
VSIM 4> # run 10
VSIM 6> # examine f_char
# A
VSIM 8> # run 10
VSIM 10> # examine f_char
# {A B}
VSIM 12> # run 10
VSIM 14> # examine f_char
# {A B C}
VSIM 16> # run 10
VSIM 18> # examine f_char
# {A B C D}
VSIM 20> # run 10
VSIM 22> # examine f_char
# {A B C D E}
VSIM 24> # run 10
VSIM 26> # examine f_char
# {B C D E}
VSIM 28> # run 10
VSIM 30> # examine f_char
# {C D E}
VSIM 32> # run 10
VSIM 34> # examine f_char
# {D E}
```

Viewing SystemC Memories

The ModelSim tool detects and displays SystemC memories. A memory is defined as any member variable of a SystemC module which is defined as an array of the following type:

- unsigned char
- unsigned short
- unsigned int
- unsigned long
- unsigned long long
- char
- short
- int
- float
- double
- enum

Properly Recognizing Derived Module Class Pointers

If you declare a pointer as a base class pointer, but actually assign a derived class object to it, then ModelSim still treats it as a base class pointer, instead of the derived class pointer, as you intended. As such, it would be unavailable for debug. To make it available for debug, you must use the **mti_set_typename** member function to instruct that it should be treated as a derived class pointer.

For instance, consider the following example:

```
class base_mod : public sc_module {

   sc_signal<int> base_sig;
   int            base_var;

   ...

};

class d1_mod : public base_mod {

   sc_signal<int> d1_sig;
   int            d1_var;
```

```
      ...

   };

   class d2_mod : public base_mod {

      sc_signal<int> d2_sig;
      int            d2_var;

      ...

   };

   SC_MODULE(top) {

       base_mod* inst;

      SC_CTOR(top) {

         if (some_condition)
            inst = new d1_mod("d1_inst");
         else
            inst = new d2_mod("d2_inst");
      }
   };
```

The **sccom** compiler can only see the declarative region of a module. Hence, in the above example, it thinks "inst" is a pointer to the "base_mod" module. After elaboration, the vsim GUI only shows "base_sig" and "base_var" in the objects window for the instance "inst".

Depending on which derived module is actually created, you really would like to see all the variables and signals of that derived class. Because we did not associate the proper derived class type with the instance "inst", the signals and variables of the derived class do exist in the kernel, however, they are not debuggable.

To correctly associate the derived class type with the instance “inst,” you can use a member function called **mti_set_typename** and apply it to the modules. You pass the actual derived class name to the function when an instance is constructed. For example,

```
   SC_MODULE(top) {

       base_mod* inst;

      SC_CTOR(top) {

         if (some_condition) {
            inst = new d1_mod("d1_inst");
            inst->mti_set_typename("d1_mod");
         } else {
            inst = new d2_mod("d2_inst");
            inst->mti_set_typename("d2_mod");
         }
      }
   };
```

In the above case, the class names happen to be simple names, which may not be true if the type is a class template with lots of template parameters. You can look up the name in *<work>/moduleinfo.sc* file, if you are unsure of the exact names.

Differences Between the Simulator and OSCI

ModelSim is based upon the 2.1 reference simulator provided by OSCI. However, there are some minor but key differences to understand:

- **vsim** calls **sc_initialize()** by default at the end of elaboration. The user has to explicitly call **sc_initialize()** in the reference simulator. You should remove calls to **sc_initialize()** from your code.
- The default time resolution of the reference simulator is 1ps. For **vsim** it is 1ns. The user can set the time resolution by using the **vsim** command with the **-t** option or by modifying the value of the Resolution variable in the *modelsim.ini* file.
- The **run** command in ModelSim is equivalent to **sc_start()**. In the reference simulator, **sc_start()** runs the simulation for the duration of time specified by its argument. In ModelSim the run command runs the simulation for the amount of time specified by its argument.
- The **sc_cycle()**, **sc_start()**, and **sc_main()** functions are not supported in ModelSim.
- The default name for **sc_object()** is bound to the actual C object name. However, this name binding only occurs after all **sc_object** constructors are executed. As a result, any **name()** function call placed inside a constructor will not pick up the actual C object name. This feature is not available on HP platforms.

Fixed-Point Types

Contrary to OSCI, ModelSim compiles the SystemC kernel with support for fixed-point types. If you want to compile your own SystemC code to enable that support, you'll need to define the compile time macro SC_INCLUDE_FX. You can do this in one of two ways:

- enter the g++/aCC argument -DSC_INCLUDE_FX on the sccom command line, such as:

```
sccom -DSC_INCLUDE_FX top.cpp
```

- add a define statement to the C++ source code before the inclusion of the *systemc.h*, as shown below:

```
#define SC_INCLUDE_FX
#include "systemc.h"
```

OSCI 2.1 Feature Implementation Details

Support for OSCI TLM Library

ModelSim includes the header files and examples from the **OSCI SystemC TLM**(Transaction Level Modeling) Library Standard version 1.0. The TLM library can be used with simulation, and requires no extra switches or files. TLM objects are not debuggable, with the exception of tlm_fifo.

Examples and documentation are located in *install_dir/examples/systemc/tlm*. The TLM header files (***tlm_*.h***) are located in *include/systemc*.

Phase Callback

The following functions are supported for phase callbacks:

- before_end_of_elaboration()
- start_of_simulation()
- end_of_simulation()

For more information regarding the use of these functions, see Initialization and Cleanup of SystemC State-Based Code.

Accessing Command-Line Arguments

The following global functions allow you to gain access to command-line arguments:

- sc_argc() — Returns the number of arguments specified on the vsim command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.
- sc_argv() — Returns the arguments specified on the vsim command line with the **-sc_arg** argument. This function can be invoked from anywhere within SystemC code.

Example:

When **vsim** is invoked with the following command line:

```
vsim -sc_arg "-a" -c -sc_arg "-b -c" -t ns -sc_arg -d
```

sc_argc() and sc_argv() will behave as follows:

```
        int argc;
        const char * const * argv;

        argc = sc_argc();
        argv = sc_argv();
```

The number of arguments (argc) is now 4.

```
argv[0] is "vsim"
argv[1] is "-a"
argv[2] is "-b -c"
argv[3] is "-d"
```

Construction Parameters for SystemC Types

The material contained in this section applies only to SystemC signals, ports , variables or fifos using one of the following fixed-point types: sc_signed, sc_unsigned, sc_fix, sc_fix_fast, sc_ufix, sc_ufix_fast. These are the only SystemC types that have construction time parameters. If you require values other than the default parameters, you need to read this section. The default size for these types is 32.

If you are using one of these types in a SystemC signal, port, fifo or an aggregate of one of these (e.g. array of sc_signal), you can not pass the size parameters to the type. This is a limitation imposed by the C++ language. Instead, SystemC provides a global default size (32) that you can control.

For sc_signed and sc_unsigned, you need to use the two objects, sc_length_param and sc_length_context, and you need to use them in an unusual way. If you just want the default vector length, simply do this:

```
SC_MODULE(dut) {
   sc_signal<sc_signed> s1;
   sc_signal<sc_signed> s2;
   SC_CTOR(dut)
   : s1("s1"), s2("s2")
   {
   }
}
```

For a single setting like using five-bit vectors, your module and its constructor would be something like:

```
SC_MODULE(dut) {
   sc_length_param l;
   sc_length_context c;
   sc_signal<sc_signed> s1;
   sc_signal<sc_signed> s2;
   SC_CTOR(dut)
```

```
      : l(5), c(l), s1("s1"), s2("s2")
      {
      }
   }
```

Notice that the constructor initialization list sets up the length parameter first, assigns the length parameter to the context object, and then constructs the two signals. You DO pass the name to the signal constructor, but the name is passed to the signal object, not to the underlying type. There is no way to reach the underlying type directly. Instead, the default constructors for sc_signed and sc_unsigned reach out to the global area and get the currently defined length parameter, the one you just set.

If you need to have signals or ports with different vector sizes, you need to include a pair of parameter and context objects for each different size:

```
SC_MODULE(dut) {
   sc_length_param l1;
   sc_length_context c1;
   sc_signal<sc_signed> s1;
   sc_signal<sc_signed> s2;

   sc_length_param l2;
   sc_length_context c2;
   sc_signal<sc_signed> u1;
   sc_signal<sc_signed> u2;
   SC_CTOR(dut)
   : l1(5), c1(l1), s1("s1"), s2("s2"),
       l2(8), c2(l2), u1("u1"), u2("u2")
   {
   }
}
```

With simple variables of this type, you reuse the context object. However, you must have the extra parameter and context objects when you are using them in a constructor-initialization list because the compiler does not allow repeated an item in that list.

The four fixed-point types that use construction parameters work exactly the same way, except that they use the objects sc_fxnum_context and sc_fxnum_params to do the work. Also, their are more parameters you can set for fixed-point numbers. Assuming we just want to set the length of the number and the number of fractional bits, here's the example above modified for fixed point numbers:

```
SC_MODULE(dut) {
   sc_fxnum_params p1;
   sc_fxnum_contxt c1;
   sc_signal<sc_fix> s1;
   sc_signal<sc_fix> s2;
   sc_fxnum_params p2;
   sc_fxnum_contxt c2;
   sc_signal<sc_ufix> u1;
   sc_signal<sc_ufix> u2;
```

```
        SC_CTOR(dut)
        : p1(5,0), c1(p1), s1("s1"), s2("s2"),
          p2(8,5), c2(p2), u1("u1"), u2("u2")
        {
        }
    }
```

Troubleshooting SystemC Errors

In the process of modifying your SystemC design to run on ModelSim, you may encounter several common errors. This section highlights some actions you can take to correct such errors.

Unexplained Behaviors During Loading or Runtime

If your SystemC simulation behaves in otherwise unexplainable ways, you should determine whether you need to adjust the stack space ModelSim allocates for threads in your design. The required size for a stack depends on the depth of functions on that stack and the number of bytes they require for automatic (local) variables.

By default the SystemC stack size is 10,000 bytes per thread.

You may have one or more threads needing a larger stack size. If so, call the SystemC function set_stack_size() and adjust the stack to accommodate your needs. Note that you can ask for too much stack space and have unexplained behavior as well.

Errors During Loading

When simulating your SystemC design, you might get a "failed to load sc lib" message because of an undefined symbol, looking something like this:

```
# Loading /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so
# ** Error: (vsim-3197) Load of
"/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so" failed: ld.so.1:
/home/icds_nut/modelsim/5.8a/sunos5/vsimk: fatal: relocation error: file
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so: symbol
_Z28host_respond_to_vhdl_requestPm:
referenced symbol not found.
# ** Error: (vsim-3676) Could not load shared library
/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so for SystemC module
'host_xtor'.
```

Source of Undefined Symbol Message

The causes for such an error could be:

- missing definition
- missing type

- using SystemC 2.1 header files from other vendors
- bad link order specified in sccom -link
- multiply-defined symbols

Missing Definition

If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

```
extern "C" void myFunc();
```

This should appear in any header files include in your C++ sources compiled by **sccom**. It tells the compiler to expect a regular C function; otherwise the compiler decorates the name for C++ and then the symbol can't be found.

Also, be sure that you actually linked with an object file that fully defines the symbol. You can use the "nm" utility on Unix platforms to test your SystemC object files and any libraries you link with your SystemC sources. For example, assume you ran the following commands:

```
sccom test.cpp
sccom -link libSupport.a
```

If there is an unresolved symbol and it is not defined in your sources, it should be correctly defined in any linked libraries:

```
nm libSupport.a | grep "mySymbol"
```

Missing Type

When you get errors during design elaboration, be sure that all the items in your SystemC design hierarchy, including parent elements, are declared in the declarative region of a module. If not, **sccom** ignores them.

For example, we have a design containing SystemC over VHDL. The following declaration of a child module "test" inside the constructor module of the code is not allowed and will produce an error:

```
SC_MODULE(Export)
{
   SC_CTOR(Export)
   {
      test *testInst;
      testInst = new test("test");
   }
};
```

The error results from the fact that the SystemC parse operation will not see any of the children of "test". Nor will any debug information be attached to it. Thus, the signal has no type information and can not be bound to the VHDL port.

The solution is to move the element declaration into the declarative region of the module.

Using SystemC 2.1 Header Files Supplied by Other Vendors

SystemC 2.1 includes version control for SystemC header files. If you compile your SystemC design using a SystemC 2.1 header file that was distributed by other vendors, and then you run **sccom -link** to link the design, an error similar to the following may result upon loading the design:

```
** Error: (vsim-3197) Load of "work/systemc.so" failed: work/systemc.so:
undefined symbol: _ZN20sc_api_version_2_1_0C1Ev.
```

To resolve the error, recompile the design using sccom. Make sure any include paths read by **sccom** do not point to a SystemC 2.1 installation. By default, **sccom** automatically picks up the ModelSim SystemC header files.

Misplaced -link Option

The order in which you place the **-link** option within the **sccom -link** command is critical. There is a big difference between the following two commands:

sccom -link liblocal.a

and

sccom liblocal.a -link

The first command ensures that your SystemC object files are seen by the linker before the library "liblocal.a" and the second command ensures that "liblocal.a" is seen first. Some linkers can look for undefined symbols in libraries that follow the undefined reference while others can look both ways. For more information on command syntax and dependencies, see sccom.

Multiple Symbol Definitions

The most common type of error found during **sccom -link** operation is the multiple symbol definition error. The error message looks something like this:

```
work/sc/gensrc/test_ringbuf.o: In function
`test_ringbuf::clock_generator(void)':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
`test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

This error arises when the same global symbol is present in more than one *.o* file. There are two common causes of this problem:

- A stale *.o* file in the working directory with conflicting symbol names.

  In this first case, just remove the stale files with the following command:

  vdel -lib <lib_path> -allsystemc

- Incorrect definition of symbols in header files.

In the second case, if you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e. not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

Text in *.h* files is included into *.cpp* files by the C++ preprocessor. By the time the compiler sees the text, it's just as if you had typed the entire text from the *.h* file into the *.cpp* file. So a *.h* file included into two *.cpp* files results in lots of duplicate text being processed by the C++ compiler when it starts up. Include guards are a common technique to avoid duplicate text problems.

If an *.h* file has an out-of-line function defined, and that *.h* file is included into two *.c* files, then the out-of-line function symbol will be defined in the two corresponding. *o* files. This leads to a multiple symbol definition error during **sccom -link**.

To solve this problem, add the "inline" keyword to give the function "internal linkage". This makes the function internal to the *.o* file, and prevents the function's symbol from colliding with a symbol in another *.o* file.

For free functions or variables, you could modify the function definition by adding the "static" keyword instead of "inline", although "inline" is better for efficiency.

Sometimes compilers do not honor the "inline" keyword. In such cases, you should move your function(s) from a header file into an out-of-line implementation in a *.cpp* file.

SystemC Procedural Interface to SystemVerilog

SystemC designs can communicate with SystemVerilog through a procedural interface, the SystemVerilog Direct Programming Interface (DPI). In contrast to a hierarchical interface, where communication is advanced through signals and ports, DPI communications consists of task and function calls passing data as arguments. This type of interface can be useful in transaction level modeling, in which bus functional models are widely used.

This section describes the use flow for using the SystemVerilog DPI to call SystemVerilog export functions from SystemC, and to call SystemC import functions from SystemVerilog.

The SystemVerilog LRM describes the details of a DPI C import and export interface. This document describes how to extend the same interface to include SystemC and C++ in general. The import and export keywords used in this document are in accordance with SystemVerilog as described in the SV LRM. An export function or task is defined in SystemVerilog, and is called by C or SystemC. An import task or function is defined in SystemC or C, and is called from SystemVerilog.

Definitions

This section defines the terms used in this section:

- C++ import function

  A C++ import function is defined as a free floating C++ function, either in the global or some private namespace. A C++ import function must not have any SystemC types as formal arguments. This function must be made available in the SystemC shared library.

- SystemC Import Function

  A SystemC import function must be available in the SystemC shared library, and can be either of the following:

  - A free floating C++ function, either in the global or private namespace, with formal arguments of SystemC types.
  - A SystemC module member function, with or without formal arguments of SystemC types.

- Export Function

  A SystemVerilog export function, as defined in the SystemVerilog LRM.

Use Flow

The use flow of SystemC DPI depends on the function modes, whether they are import or export. The import and export calls described in the following sections can be mixed with each other in any order.

SystemC Import Functions

In order to make a SystemC import function callable from SystemVerilog, it needs to be registered from the SystemC code before it can be called from SystemVerilog. This can be thought of as exporting the function outside SystemC, thus making it callable from other languages. The registration must be done by passing a pointer to the function using an API. The registration can be done anywhere in the design but it must be done before the call happens in SystemVerilog, otherwise the call fails with undefined behavior.

Global Functions

A global function can be registered using the API below:

```
int sc_dpi_register_cpp_function(const char* function_name, PTFN
func_ptr);
```

This function takes two arguments:

- the name of the function, which can be different than the actual function name. This name must match the SystemVerilog import declaration. No two function registered using this API can have the same name: it creates an error if they do.
- a function pointer to the registered function. On successful registration, this function will return a 0. A non-zero return status means an error.

Example 8-4. Global Import Function Registration

```
int scGlobalImport(sc_logic a, sc_lv<9>* b);
sc_dpi_register_cpp_function(“scGlobalImport”, scGlobalImport);
```

A macro like the one shown below is provided to make the registration even more simple. In this case the ASCII name of the function will be identical to the name of the function in the source code.

```
SC_DPI_REGISTER_CPP_FUNCTION(scGlobalImport);
```

In the SystemVerilog code, the import function needs to be defined with a special marker ("DPI-SC") that tells the SV compiler that this is an import function defined in the SystemC shared library. The syntax for calling the import function remains the same as described in the SystemVerilog LRM.

Example 8-5. SystemVerilog Global Import Declaration

For the SystemC import function shown in Example 8-4, the SystemVerilog import declaration is as follows:

```
import mti_scdpi::*;
import "DPI-SC" context function int scGlobalImport(
                        input sc_logic a, output sc_lv[8:0] b);
```

Please refer to the sections below for more details on the SystemC import and export task or function declaration syntax.

Module Member Functions

Registering Functions

Module member functions can be registered anytime before they are called from the SystemVerilog code. The following macro can be used to register a non-static member function if the registration is done from a module constructor or a module member function. For a static member function, the registration is accomplished using the interface SC_DPI_REGISTER_CPP_FUNCTION, as described in SystemC Import Functions.

```
SC_DPI_REGISTER_CPP_MEMBER_FUNCTION(<function_name>, <func_ptr>);

Example:

SC_MODULE(top) {

    void sc_func() {
    }

    SC_CTOR(top) {
        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION(“sc_func”, &top::sc_func);
    }
};
```

Note that in the above case, since the registration is done from the module constructor, the module pointer argument might be redundant. However, the module pointer argument will be required if the macro is used outside a constructor.

To register a member function from a function that is not a member of the module, the following registration function must be used:

```
int sc_dpi_register_cpp_member_function(<function_name>, <module_ptr>,
<func_ptr>);
```

This function takes three arguments. The first argument is the name of the function, which can be different than the actual function name. This is the name that must be used in the SystemVerilog import declaration. The second argument is a reference to the module instance where the function is defined. It is illegal to pass a reference to a class other than a class derived from sc_module and will lead to undefined behavior. The third and final argument is a function pointer to the member function being registered. On successful registration, this function will return a 0. A non-zero return status means an error. For example, the member function run() of the module "top" in the example above can be registered as follows:

```
sc_module* pTop = new top("top");
sc_dpi_register_cpp_member_function("run", pTop, &top::run);
```

Setting Stack Size for Import Tasks

The tool implicitly creates a SystemC thread to execute the C++ functions declared as SystemVerilog import tasks. The default stack size is 64KBytes and may not be big enough for any C++ functions. To change the default stack size, you can use the interface "sc_dpi_set_stack_size". You must use this interface right after the registration routine, for example:

```
SC_MODULE(top) {

   void sc_task() {
   }

   SC_CTOR(top) {
      SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_task", &top::sc_task);
      sc_dpi_set_stack_size(1000000);     // set stack size to be 1Mbyte.
      }
   }
```

For the C++ functions declared as SystemVerilog import functions, you do not need to set the stack size.

Declaring and Calling Member Import Functions in SystemVerilog

The declaration for a member import function in SystemVerilog is similar to the following:

```
   import "DPI-SC" context function int scMemberImport(
                           input sc_logic a, output sc_lv[8:0] b);
```

Registration of static member functions is identical to the registration of global functions using the API sc_dpi_register_cpp_function().

Only one copy of the overloaded member functions is supported as a DPI import, as DPI can only identify the import function by its name. not by the function parameters.

To enable the registration of member functions, the SystemC source file must be compiled with the “-DMTI_BIND_SC_MEMBER_FUNCTION” macro.

Calling SystemVerilog Export Tasks / Functions from SystemC

Unless an export call is made from an import function, you must set the scope of the export function explicitly to provide the SystemVerilog context information to the simulator. You do this by calling svSetScope() before each export function or task call.

An export function to be called with SystemC arguments must have an export declaration, similar to the following:

```
   export "DPI-SC" context function Export;
```

The function declaration must use the SystemC type package, similar to the following:

```
   import mti_scdpi::*;
   function int Export(input sc_logic a, output sc_bit b);
```

The syntax for calling an export function from SystemC is the same as any other C++ function call.

SystemC Data Type Support in SystemVerilog DPI

The SystemVerilog package “scdpi” must be imported if a SystemC data type is used in the arguments of import and export functions.

```
   import mti_scdpi::*
```

The SystemC data type names have been treated as special keywords. Avoid using these keywords for other purposes in your SystemVerilog source files.

The table below shows how each of the SystemC type will be represented in SystemVerilog. This table must be followed strictly for passing arguments of SystemC type. The SystemVerilog typedef statements, listed in the middle column of Table 8-9, are automatically imported whenever the mti_scdpi package is imported.

Table 8-9. SystemC Types as Represented in SystemVerilog

SystemC Type	SV Typedef	Import/Export Declaration
sc_logic	typedef logic sc_logic	sc_logic
sc_bit	typedef bit sc_bit	sc_bit
sc_bv<N>	typedef bit sc_bv	sc_bit[N-1:0]
sc_lv<N>	typedef logic sc_lv	sc_lv[N-1:0]
sc_int<N>	typedef bit sc_int	sc_int[N-1:0]
sc_uint<N>	typedef bit sc_uint	sc_uint[N-1:0]
sc_bigint<N>	typedef bit sc_bigint	sc_bigint[N-1:0]
sc_biguint<N>	typedef bit sc_biguint	sc_biguint[N-1:0]
sc_fixed<W,I,Q,O,N>	typedef bit sc_fixed	sc_fixed[I-1:I-W]
sc_ufixed<W,I,Q,O,N>	typedef bit sc_ufixed	sc_ufixed[I-1:I-W]
sc_fixed_fast<W...>	typedef bit sc_fixed_fast	sc_fixed[I-1:I-W]
sc_ufixed_fast<W...>	typedef bit sc_ufixed_fast	sc_fixed[I-1:I-W]
sc_signed	typedef bit sc_signed	sc_signed[N-1:0]
sc_unsigned	typedef bit sc_unsigned	sc_unsigned[N-1:0]
sc_fix	typedef bit sc_fix	sc_fix[I-1:1-W]
sc_ufix	typedef bit sc_ufix	sc_ufix[I-1:1-W]
sc_fix_fast	typedef bit sc_fix_fast	sc_fix_fast[I-1:1-W]
sc_ufix_fast	typedef bit sc_ufix_fast	sc_ufix_fast[I-1:1-W]

According to the table above, a SystemC argument of type **sc_uint<32>** will be declared as **sc_uint[31:0]** in SystemVerilog “DPI-SC” declaration. Similarly, **sc_lv<9>** would be **sc_lv[8:0]**. to enable the fixed point datatypes, the SystemC source file must be compiled with -DSC_INCLUDE_FX.

For fixed-point types the left and right indexes of the SV vector can lead to a negative number. For example, **sc_fixed<3,0>** will translate to **sc_fixed[0-1:0-3]** which is **sc_fixed[-1:-3]**. This representation is used for fixed-point numbers in the ModelSim tool, and must be strictly followed.

For the SystemC types whose size is determined during elaboration, such as sc_signed and sc_unsigned, a parameterized array must be used on the SV side. The array size parameter value, on the SystemVerilog side, must match correctly with the constructor arguments passed to types such as sc_signed and sc_unsigned at SystemC elaboration time.

Some examples:

An export declaration with arguments of SystemC type:

```
export "DPI-SC" context function Export;

import mti_scdpi::*;
function int Export(input sc_logic a, input sc_int[8:0] b);
```

An import function with arguments of SystemC type:

```
import mti_scdpi::*;
import "DPI-SC" context function int scGlobalImport(
                        input sc_logic a, output sc_lv[8:0] b);
```

An export function with arguments of regular C types:

```
export "DPI-SC" context function Export;
function int Export(input int a, output int b);
```

SystemC Function Prototype Header File (sc_dpiheader.h)

The sc_dpiheader.h file contains the C function prototype statements consistent with the "DPI-SC" import/export function/task declaration. It is generated by vlog automatically, when compiling Verilog source files. You can use this file as a sanity check for the SystemC function arguments and return type declaration in your source files.

SystemC DPI Usage Example

```
----------------------------------------
hello.v:


module top;

hello c_hello();

import "DPI-SC" context function void sc_func();
export "DPI-SC" task verilog_task;

task verilog_task();
  $display("hello from verilog_task.");
endtask

initial
begin

  sc_func();

  #2000 $finish;
end
endmodule
```

```
----------------------------------------

hello.cpp:

#include "systemc.h"
#include "sc_dpiheader.h"

SC_MODULE(hello)
{
    void call_verilog_task();
    void sc_func();

    SC_CTOR(hello)
    {
        SC_THREAD(call_verilog_task);

        SC_DPI_REGISTER_CPP_MEMBER_FUNCTION("sc_func", &hello::sc_func);
    }

    ~hello() {};
};

void hello::sc_func()
{
    printf("hello from sc_func().

}

void hello::call_verilog_task()
{
    svSetScope(svGetScopeFromName("top"));

    for(int i = 0; i < 3; ++i)
    {
        verilog_task();
    }
}

SC_MODULE_EXPORT(hello);


----------------------------------------
Compilation:


vlog -sv hello.v

sccom -DMTI_BIND_SC_MEMBER_FUNCTION hello.cpp

sccom -link

vsim -c -do "run -all; quit -f" top
```

Chapter 9
Mixed-Language Simulation

ModelSim single-kernel simulation allows you to simulate designs that are written in VHDL, Verilog, SystemVerilog, and SystemC. The boundaries between languages are enforced at the level of a design unit. This means that although a design unit itself must be entirely of one language type, it may instantiate design units from another language. Any instance in the design hierarchy may be a design unit from another language without restriction.

Basic Mixed-Language Flow

Simulating mixed-language designs with ModelSim includes these general steps:

1. Compile HDL source code using vcom or vlog. Compile SystemC C++ source code using sccom. Compile all modules in the design following order-of-compile rules.
   - For SystemC designs with HDL instances — Create a SystemC foreign module declaration for all Verilog and VHDL instances (see SystemC Foreign Module (Verilog) Declaration or SystemC Foreign Module (VHDL) Declaration).
   - For Verilog/VHDL designs with SystemC instances — Export any SystemC instances that will be directly instantiated by Verilog/VHDL using the SC_MODULE_EXPORT macro. Exported SystemC modules can be instantiated just as you would instantiate any Verilog/VHDL module or design unit.
   - For binding Verilog design units to VHDL or Verilog design units — See “Using the SystemVerilog bind Construct in Mixed-Language Designs.” When using bind in compilation unit scope, use the **-cuname** argument with the vlog command (see Handling Bind Statements in the Compilation Unit Scope).
2. For designs containing SystemC — Link all objects in the design using sccom -link.
3. Elaborate and optimize your design using the vopt command. See Optimizing Mixed Designs.
4. Simulate the design with the vsim command**.**
5. Run and debug your design.

Separate Compilers with Common Design Libraries

VHDL source code is compiled by vcom and the resulting compiled design units (entities, architectures, configurations, and packages) are stored in the working library. Likewise, Verilog source code is compiled by vlog and the resulting design units (modules and UDPs) are stored in the working library.

SystemC/C++ source code is compiled with the sccom command. The resulting object code is compiled into the working library.

Design libraries can store any combination of design units from any of the supported languages, provided the design unit names do not overlap (VHDL design unit names are changed to lower case). See Design Libraries for more information about library management.

Access Limitations in Mixed-Language Designs

The Verilog language allows hierarchical access to objects throughout the design. This is not the case with VHDL or SystemC. You *cannot* directly read or change a VHDL or SystemC object (signal, variable, generic, etc.) with a hierarchical reference within a mixed-language design. Furthermore, you cannot directly access a Verilog object up or down the hierarchy if there is an interceding VHDL or SystemC block.

You have two options for accessing VHDL objects or Verilog objects "obstructed" by an interceding block: 1) propagate the value through the ports of all design units in the hierarchy; 2) use the Signal Spy procedures or system tasks (see Signal Spy for details).

To access obstructed SystemC objects, propagate the value through the ports of all design units in the hierarchy or use the control/observe functions. The following two member functions of sc_signal may be used to control and observe hierarchical signals in a design:

- control_foreign_signal()
- observe_foreign_signal()

For more information on the use of control and observe, see Hierarchical References In Mixed HDL/SystemC Designs.

Using the SystemVerilog bind Construct in Mixed-Language Designs

The SystemVerilog **bind** construct allows you to bind a Verilog design unit to another Verilog design unit or to a VHDL design unit. This is especially useful for binding SystemVerilog assertions to your VHDL, Verilog and mixed designs during verification.

Binding one design unit to another is a simple process of creating a module that you want to bind to a target design unit, then writing a bind statement. For example, if you want to bind a SystemVerilog assertion module to a VHDL design you would do the following:

1. Write assertions inside a Verilog module.

2. Designate a target VHDL entity or a VHDL entity/architecture pair.
3. Bind the assertion module to the target with a **bind** statement.

Modules, programs, or interfaces can be bound to:

- all instances of a target module
- a specific instance of the target module
- all instances that use a certain architecture in the target module

Binding to a configuration is not allowed.

Syntax of bind Statement

```
bind <target_entity/architecture_name> <assertion_module_name>
<instance_name> <port connections>
```

This **bind** statement will create an instance of the assertion module inside the target VHDL entity/architecture with the specified instance name and port connections. When the target is a VHDL entity, the bind instance is created under the last compiled architecture. It should be noted that the instance that is being bound cannot contain another bind statement. In addition, a bound instance can make hierarchical reference into the design but the design cannot make hierarchical reference inside a bound instance.

Binding to VHDL Enumerated Types

SystemVerilog infers an enumeration concept similar to VHDL enumerated types. In VHDL, the enumerated names are assigned to a fixed enumerated value, starting left-most with the value 0. In SystemVerilog, you can also explicitly define the enumerated values. As a result, the bind construct can be use for port mapping of VHDL enumerated types to Verilog port vectors.

The actual expression in a bind port map must be simple names (including hierarchical names if the target is a Verilog design unit) and Verilog literals. For example:

```
bind target checker inst(req, ack, 1;b1)
```

is a legal expression; whereas,

```
bind target checker inst(req | ack, {req1, req2})
```

is illegal because the actual expressions are neither simple names nor literals.

Port mapping is supported for both input and output ports. The integer value of the enum is first converted to a bit vector before connecting to a Verilog formal port. Note that you cannot connect enum value on port actual – it has to be signal. In addition, port vectors can be of any size less than or equal to 32.

This kind of port mapping between VHDL enum and Verilog vector is only allowed when the Verilog is instantiated under VHDL through the bind construct and is not supported for normal instances.

The allowed VHDL types for port mapping to SystemVerilog port vectors are:

bit	std_logic	vl_logic
bit_vector	std_logic_vector	vl_logic_vector

Example of Binding to VHDL Enumerated Types

Suppose you want to use SVA to monitor a VHDL finite state machine that uses enumerated types. With Questa, you can map VHDL enumerated types to Verilog integer or real types. This enables binding to VHDL enumerated types. Consider the following VHDL code:

```
...
   type fsm_state is(idle, send_bypass,
      load0,send0, load1,send1, load2,send2,
      load3,send3, load4,send4, load5,send5,
      load6,send6, load7,send7, load8,send8,
      load9,send9, load10,send10,
      load_bypass, wait_idle);
   signal int_state : fsm_state;
   signal nxt_state : fsm_state;
...
```

First you define the properties to be monitored in the SystemVerilog module. Then you map the vector to the enumerated name. Because fsm_state has 26 values, you need a 5-bit vector for an input port:

```
typedef enum {idle, send_bypass,
      load0,send0, load1,send1, load2,send2,
      load3,send3, load4,send4, load5,send5,
      load6,send6, load7,send7, load8,send8,
      load9,send9, load10,send10,
      load_bypass, wait_idle} fsm_state;
module interleaver_props (
   input clk, in_hs, out_hs,
   input [4:0] int_state_vec
);

fsm_state int_state;

assign int_state = int_state_vec; // Map vector to enum name
...
// Check for sync byte at the start of a every packet property
pkt_start_check;
- @(posedge clk) (int_state == idle && in_hs) -> (sync_in_valid);
endproperty
...
```

Handling Bind Statements in the Compilation Unit Scope

Bind statements are allowed in module, interface, and program blocks, and may exist in the compilation unit scope. ModelSim treats the compilation unit scope ($unit) as a package – internally wrapping the content of $unit into a package.

Before vsim elaborates a module it elaborates all packages upon which that module depends. In other words, it elaborates a $unit package before a module in the compilation unit scope.

It should be noted that when the **bind** statement is in the compilation unit scope, the **bind** only becomes effective when $unit package gets elaborated by vsim. In addition, the package gets elaborated only when a design unit that depends on that package gets elaborated. So if you have a file in a compilation unit scope that contains only **bind** statements, you can compile that file by itself, but the **bind** statements will never be elaborated. A warning to this effect is generated by vlog if **bind** statements are found in the compilation unit scope.

The **-cuname** argument for vlog gives a user-defined name to a specified compilation $unit package (which, in the absence of **-cuname**, is some implicitly generated name). You must provide this named compilation unit package with the vsim command as the top level design unit in order to force elaboration.

The **-cuname** argument is used only in conjunction with the **-mfcu** argument, which instructs the compiler to treat all files within a compilation command line as a single compilation unit.

Example

Suppose you have a SystemVerilog module, called *checker.sv*, that contains an assertion for checking a counter:

```
module checker(clk, reset, cnt);
parameter SIZE = 4;
input clk;
input reset;
input [SIZE-1:0] cnt;
property check_count;
   @(posedge clk)
   !reset |=> cnt == ($past(cnt) + 1);
endproperty
assert property (check_count);
endmodule
```

You want to **bind** that to a counter module named *counter.sv*.

```
module counter(clk, reset, cnt);
parameter SIZE = 8;
input clk;
input reset;
output [SIZE-1:0] cnt;
reg [SIZE-1:0] cnt;
always @(posedge clk)
```

```
begin
   if (reset == 1'b1)
      cnt = 0;
   else
      cnt = cnt + 1;
end
endmodule
```

The **bind** statement is in a file named *bind.sv*, which will reside in the compilation unit scope.

```
bind counter checker #(SIZE) checker_inst(clk, reset, cnt);
```

This statement instructs ModelSim to create an instance of *checker* in the target module, *counter.sv*.

The final component of this design is a testbench, named *tb.sv*.

```
module testbench;
reg clk, reset;
wire [15:0] cnt;
counter #(16) inst(clk, reset, cnt);
initial
begin
   clk = 1'b0;
   reset = 1'b1;
   #500 reset = 1'b0;
   #1000 $finish;
end
always #50 clk = ~clk;
endmodule
```

If the bind.sv file is compiled by itself (vlog bind.sv), you will receive a Warning like this one:

```
** Warning: 'bind' found in a compilation unit scope that either does not
contain any design units or only contains design units that are
instantiated by 'bind'. The 'bind' instance will not be elaborated.
```

To fix this problem, use the -cuname argument with vlog as follows:

```
vlog -cuname bind_pkg -mfcu bind.sv
```

Then simulate the design with:

```
vsim testbench bind_pkg
```

If you are using the **vlog -R** or qverilog commands to compile and simulate the design, this binding issue is handled properly automatically.

Optimizing Mixed Designs

The vopt command performs global optimizations to improve simulator performance. You run **vopt** on the top-level design unit. See Optimizing Designs with vopt for further details.

Simulator Resolution Limit

In a mixed-language design with only one top, the resolution of the top design unit is applied to the whole design. If the root of the mixed design is VHDL, then VHDL simulator resolution rules are used (see Simulator Resolution Limit (VHDL) for VHDL details). If the root of the mixed design is Verilog, Verilog rules are used (see Simulator Resolution Limit (Verilog) for Verilog details). If the root is SystemC, then SystemC rules are used (see SystemC Time Unit and Simulator Resolution for SystemC details).

In the case of a mixed-language design with multiple tops, the following algorithm is used:

- If VHDL or SystemC modules are present, then the Verilog resolution is ignored. An error is issued if the Verilog resolution is finer than the chosen one.
- If both VHDL and SystemC are present, then the resolution is chosen based on which design unit is elaborated first. For example:

  ```
  vsim sc_top vhdl_top -do vsim.do
  ```

  In this case, the SystemC resolution (default 1 ns) is chosen.

  ```
  vsim vhdl_top sc_top -do vsim.do
  ```

  In this case, the VHDL resolution is chosen.
- All resolutions specified in the source files are ignored if **vsim** is invoked with the **-t** option. When set, this overrides all other resolutions.

Runtime Modeling Semantics

The ModelSim simulator is compliant with all pertinent Language Reference Manuals. To achieve this compliance, the sequence of operations in one simulation iteration (i.e. delta cycle) is as follows:

- SystemC processes are run
- Signal updates are made
- HDL processes are run

The above scheduling semantics are required to satisfy both the SystemC and the HDL LRM. Namely, all processes triggered by an event in a SystemC primitive channel shall wake up at the beginning of the following delta. All processes triggered by an event on an HDL signal shall wake up at the end of the current delta. For a signal chain that crosses the language boundary, this means that processes on the SystemC side get woken up one delta later than processes on the HDL side. Consequently, one delta of skew will be introduced between such processes. However, if the processes are communicating with each other, correct system behavior will still result.

Hierarchical References In Mixed HDL/SystemC Designs

A SystemC signal (including sc_signal, sc_buffer, sc_signal_resolved, and sc_signal_rv) can control or observe an HDL signal using two member functions of sc_signal:

```
bool control_foreign_signal(const char* name);
bool observe_foreign_signal(const char* name);
```

The argument (const char* name) is a full hierarchical path to an HDL signal or port. The return value is "true" if the HDL signal is found and its type is compatible with the SystemC signal type. See tables for Verilog Data Type Mapping to Verilog and VHDL Data Type Mapping to VHDL to view a list of types supported at the mixed language boundary. If it is a supported boundary type, it is supported for hierarchical references. If the function is called during elaboration time, when the HDL signal has not yet elaborated, the function always returns "true"; however, an error is issued before simulation starts.

Control

When a SystemC signal calls **control_foreign_signal()** on an HDL signal, the HDL signal is considered a fanout of the SystemC signal. This means that every value change of the SystemC signal is propagated to the HDL signal. If there is a pre-existing driver on the HDL signal which has been controlled, the value is changed to reflect the SystemC signal’s value. This value remains in effect until a subsequent driver transaction occurs on the HDL signal, following the semantics of the **force -deposit** command.

Observe

When a SystemC signal calls **observe_foreign_signal()** on an HDL signal, the SystemC signal is considered a fanout of the HDL signal. This means that every value change of the HDL signal is propagated to the SystemC signal. If there is a pre-existing driver on the SystemC signal which has been observed, the value is changed to reflect the HDL signal’s value. This value remains in effect until a subsequent driver transaction occurs on the SystemC signal, following the semantics of the **force -deposit** command.

Once a SystemC signal executes a control or observe on an HDL signal, the effect stays throughout the whole simulation. Any subsequent control/observe on that signal will be an error.

Example:

```
SC_MODULE(test_ringbuf)
{
    sc_signal<bool> observe_sig;
    sc_signal<sc_lv<4> > control_sig;

    // HDL module instance
    ringbuf* ring_INST;

    SC_CTOR(test_ringbuf)
    {
        ring_INST = new ringbuf("ring_INST", "ringbuf");
        .....

observe_sig.observe_foreign_signal("/test_ringbuf/ring_INST/block1_INST/b
uffers(0)");

control_sig.control_foreign_signal("/test_ringbuf/ring_INST/block1_INST/s
ig");
    }
};
```

Mapping Data Types

Cross-language (HDL) instantiation does not require any extra effort on your part. As ModelSim loads a design it detects cross-language instantiations – made possible because a design unit's language type can be determined as it is loaded from a library – and the necessary adaptations and data type conversions are performed automatically. SystemC and HDL cross-language instantiation requires minor modification of SystemC source code (addition of SC_MODULE_EXPORT, sc_foreign_module, etc.).

A VHDL instantiation of Verilog may associate VHDL signals and values with Verilog ports and parameters. Likewise, a Verilog instantiation of VHDL may associate Verilog nets and values with VHDL ports and generics. The same holds true for SystemC and VHDL/Verilog ports.

ModelSim automatically maps between the language data types as shown in the sections below.

Verilog to VHDL Mappings

Verilog Parameters

The type of a Verilog parameter is determined by its initial value.

Verilog type	VHDL type
integer	integer
real	real
string	string

Verilog Ports

The allowed VHDL types for ports connected to Verilog nets and for signals connected to Verilog ports are:

Allowed VHDL types
bit
bit_vector
std_logic
std_logic_vector
vl_logic
vl_logic_vector

The vl_logic type is an enumeration that defines the full state set for Verilog nets, including ambiguous strengths. The bit and std_logic types are convenient for most applications, but the vl_logic type is provided in case you need access to the full Verilog state set. For example, you may wish to convert between vl_logic and your own user-defined type. The vl_logic type is defined in the vl_types package in the pre-compiled **verilog** library. This library is provided in the installation directory along with the other pre-compiled libraries (**std** and **ieee**). The source code for the vl_types package can be found in the files installed with ModelSim. (See *<install_dir>/modeltech/vhdl_src/verilog/vltypes.vhd*.)

Verilog States

Verilog states are mapped to std_logic and bit as follows:

Verilog	std_logic	bit	Verilog	std_logic	bit
HiZ	'Z'	'0'	Pu0	'L'	'0'
Sm0	'L'	'0'	Pu1	'H'	'1'
Sm1	'H'	'1'	PuX	'W'	'0'
SmX	'W'	'0'	St0	'0'	'0'
Me0	'L'	'0'	St1	'1'	'1'
Me1	'H'	'1'	StX	'X'	'0'
MeX	'W'	'0'	Su0	'0'	'0'
We0	'L'	'0'	Su1	'1'	'1'
We1	'H'	'1'	SuX	'X'	'0'
WeX	'W'	'0'			
La0	'L'	'0'			

Verilog	std_logic	bit	Verilog	std_logic	bit
La1	'H'	'1'			
LaX	'W'	'0'			

For Verilog states with ambiguous strength:

- bit receives '0'
- std_logic receives 'X' if either the 0 or 1 strength component is greater than or equal to strong strength
- std_logic receives 'W' if both the 0 and 1 strength components are less than strong strength

VHDL To Verilog Mappings

VHDL Generics

VHDL type	Verilog type
integer	integer or real
real	integer or real
time	integer or real
physical	integer or real
enumeration	integer or real
string	string literal

When a scalar type receives a real value, the real is converted to an integer by truncating the decimal portion.

Type time is treated specially: the Verilog number is converted to a time value according to the **‘timescale** directive of the module.

Physical and enumeration types receive a value that corresponds to the position number indicated by the Verilog number. In VHDL this is equivalent to T'VAL(P), where T is the type, VAL is the predefined function attribute that returns a value given a position number, and P is the position number.

VHDL type bit is mapped to Verilog states as follows:

bit	Verilog
'0'	St0
'1'	St1

VHDL type std_logic is mapped to Verilog states as follows:

std_logic	Verilog
'U'	StX
'X'	StX
'0'	St0
'1'	St1
'Z'	HiZ
'W'	PuX
'L'	Pu0
'H'	Pu1
'–'	StX

Verilog And SystemC Signal Interaction And Mappings

SystemC has a more complex signal-level interconnect scheme than Verilog. Design units are interconnected via hierarchical and primitive channels. An sc_signal<> is one type of primitive channel. The following section discusses how various SystemC channel types map to Verilog wires when connected to each other across the language boundary.

Channel and Port Type Mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to Verilog modules).

Channels	Ports	Verilog mapping
sc_signal<T>	sc_in<T> sc_out<T>sc_inout<T>	Depends on type T. See table entitled Data Type Mapping to Verilog.
sc_signal_rv<W>	sc_in_rv<W> sc_out_rv<W> sc_inout_rv<W>	wire [W-1:0]
sc_signal_resolved	sc_in_resolved sc_out_resolved sc_inout_resolved	wire [W-1:0]
sc_clock	sc_in_clk sc_out_clk sc_inout_clk	wire

Channels	Ports	Verilog mapping
sc_mutex	N/A	Not supported on language boundary
sc_fifo	sc_fifo_in sc_fifo_out	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary[1]

[1] User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.

- A SystemC sc_out port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A read() operation on such an sc_out port might give incorrect values. Use an sc_inout port to do both read() and write() operations.

Data Type Mapping to Verilog

SystemC's sc_signal<> types are mapped to Verilog types as follows:

SystemC	Verilog
bool, sc_bit	wire
sc_logic	wire
sc_bv<W>	wire [W-1:0]
sc_lv<W>	wire [W-1:0]
sc_int<W>, sc_uint<W>	wire [W-1:0]
sc_bigint<W>, sc_biguint<W>	wire [W-1:0]
sc_fixed<W,I,Q,O,N>, sc_ufixed<W,I,Q,O,N>	wire [W-1:0]
sc_fixed_fast<W,I,Q,O,N>, sc_ufixed_fast<W,I,Q,O,N>	wire [W-1:0]
[1]sc_fix, [1]sc_ufix	wire [WL-1:0]

SystemC	Verilog
[1]sc_fix_fast, [1]sc_ufix_fast	wire [WL-1:0]
[2]sc_signed, [2]sc_unsigned	wire[WL-1:0]
char, unsigned char	wire [7:0]
short, unsigned short	wire [15:0]
int, unsigned int	wire [31:0]
long, unsigned long	wire [31:0]
long long, unsigned long long	wire [63:0]
float	wire [31:0]
double	wire [63:0]
enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
struct	Not supported on language boundary
union	Not supported on language boundary
bit_fields	Not supported on language boundary

1. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type sc_fix, sc_ufix, sc_fix_fast, or sc_ufix_fast of word length other than the default(32), you must use sc_fxtype_params and sc_fxtype_context to set the word length. For more information, see Construction Parameters for SystemC Types.
2. To make a port of type sc_signed or sc_unsigned of word length other than the default (32), you must use sc_length_param and sc_length_context to set the word length. For more information, see Construction Parameters for SystemC Types.

Port Direction

Verilog port directions are mapped to SystemC as follows:

Verilog	SystemC
input	sc_in<T>, sc_in_resolved, sc_in_rv<W>
output	sc_out<T>, sc_out_resolved, sc_out_rv<W>
inout	sc_inout<T>, sc_inout_resolved, sc_inout_rv<W>

Verilog to SystemC State Mappings

Verilog states are mapped to sc_logic, sc_bit, and bool as follows:

Verilog	**sc_logic**	**sc_bit**	**bool**
HiZ	'Z'	'0'	false
Sm0	'0'	'0'	false
Sm1	'1'	'1'	true
SmX	'X'	'0'	false
Me0	'0'	'0'	false
Me1	'1'	'1'	true
MeX	'X'	'0'	false
We0	'0'	'0'	false
We1	'1'	'1'	true
WeX	'X'	'0'	false
La0	'0'	'0'	false
La1	'1'	'1'	true
LaX	'X'	'0'	false
Pu0	'0'	'0'	false
Pu1	'1'	'1'	true
PuX	'X'	'0'	false
St0	'0'	'0'	false
St1	'1'	'1'	true
StX	'X'	'0'	false
Su0	'0'	'0'	false
Su1	'1'	'1'	true
SuX	'X'	'0'	false

For Verilog states with ambiguous strength:

- sc_bit receives '1' if the value component is 1, else it receives ’0’
- bool receives true if the value component is 1, else it receives false
- sc_logic receives 'X' if the value component is X, H, or L
- sc_logic receives '0' if the value component is 0

- sc_logic receives ’1’ if the value component is 1

SystemC to Verilog State Mappings

SystemC type bool is mapped to Verilog states as follows:

bool	**Verilog**
false	St0
true	St1

SystemC type sc_bit is mapped to Verilog states as follows:

sc_bit	**Verilog**
'0'	St0
'1'	St1

SystemC type sc_logic is mapped to Verilog states as follows:

sc_logic	**Verilog**
'0'	St0
'1'	St1
'Z'	HiZ
'X'	StX

VHDL and SystemC Signal Interaction And Mappings

SystemC has a more complex signal-level interconnect scheme than VHDL. Design units are interconnected via hierarchical and primitive channels. An sc_signal<> is one type of primitive channel. The following section discusses how various SystemC channel types map to VHDL types when connected to each other across the language boundary.

Port Type Mapping

The following port type mapping table lists all channels. Three types of primitive channels and one hierarchical channel are supported on the language boundary (SystemC modules connected to VHDL modules).

Channels	**Ports**	**VHDL mapping**
sc_signal<T>	sc_in<T> sc_out<T> sc_inout<T>	Depends on type T. See table entitled Data Type Mapping to VHDL.

Channels	Ports	VHDL mapping
sc_signal_rv<W>	sc_in_rv<W> sc_out_rv<W> sc_inout_rv<W>	std_logic_vector(W-1 downto 0)
sc_signal_resolved	sc_in_resolved sc_out_resolved sc_inout_resolved	std_logic
sc_clock	sc_in_clk sc_out_clk sc_inout_clk	bit/std_logic/boolean
sc_mutex	N/A	Not supported on language boundary
sc_fifo	sc_fifo_in sc_fifo_out	Not supported on language boundary
sc_semaphore	N/A	Not supported on language boundary
sc_buffer	N/A	Not supported on language boundary
user-defined	user-defined	Not supported on language boundary[1]

[1] User defined SystemC channels and ports derived from built-in SystemC primitive channels and ports can be connected to HDL signals. The built-in SystemC primitive channel or port must be already supported at the mixed-language boundary for the derived class connection to work.

- A SystemC sc_out port connected to an HDL signal higher up in the design hierarchy is treated as a pure output port. A read() operation on such an sc_out port might give incorrect values. Use an sc_inout port to do both read() and write() operations.

Data Type Mapping to VHDL

SystemC's sc_signal types are mapped to VHDL types as follows

SystemC	VHDL
bool, sc_bit	bit/std_logic/boolean
sc_logic	std_logic
sc_bv<W>	bit_vector(W-1 downto 0)
sc_lv<W>	std_logic_vector(W-1 downto 0)
sc_bv<32>, sc_lv<32>	integer

SystemC	VHDL
sc_bv<64>, sc_lv<64>	real

sc_int<W>,
sc_uint<W>
	bit_vector(W-1 downto 0)
	std_logic_vector(W -1 downto 0)

sc_bigint<W>,
sc_biguint<W>
	bit_vector(W-1 downto 0)
	std_logic_vector(W-1 downto 0)

sc_fixed<W,I,Q,O,N>,
sc_ufixed<W,I,Q,O,N>
	bit_vector(W-1 downto 0)
	std_logic_vector(W-1 downto 0)

sc_fixed_fast<W,I,Q,O,N>,
sc_ufixed_fast<W,I,Q,O,N>
	bit_vector(W-1 downto 0)
	std_logic_vector(W-1 downto 0)

[1]sc_fix,
[1]sc_ufix
	bit_vector(WL-1 downto 0)
	std_logic_vector(WL- 1 downto 0)

[1]sc_fix_fast,
[1]sc_ufix_fast
	bit_vector(WL-1 downto 0)
	std_logic_vector(WL- 1 downto 0)

[2]sc_signed,
[2]sc_unsigned
	bit_vector(WL-1 downto 0)
	std_logic_vector(WL- 1 downto 0)

char, unsigned char	bit_vector(7 downto 0)
	std_logic_vector(7 downto 0)

short, unsigned short	bit_vector(15 downto 0)
	std_logic_vector(15 downto 0)

int, unsigned int	bit_vector(31 downto 0)
	std_logic_vector(7 downto 0)

long, unsigned long	bit_vector(31 downto 0)
	std_logic_vector(31 downto 0)

long long, unsigned long long	bit_vector(63 downto 0)
	std_logic_vector(63 downto 0)

float	bit_vector(31 downto 0)
	std_logic_vector(31 downto 0)

double	bit_vector(63 downto 0)
	std_logic_vector(63 downto 0)
	real

enum	Not supported on language boundary
pointers	Not supported on language boundary
class	Not supported on language boundary
structure	Not supported on language boundary

SystemC	VHDL
union	Not supported on language boundary
bit_fields	Not supported on language boundary

1. WL (word length) is the total number of bits used in the type. It is specified during runtime. To make a port of type sc_fix, sc_ufix, sc_fix_fast, or sc_ufix_fast of word length other than the default(32), you must use sc_fxtype_params and sc_fxtype_context to set the word length. For more information, see Construction Parameters for SystemC Types.
2. To make a port of type sc_signed or sc_unsigned of word length other than the default (32), you must use sc_length_param and sc_length_context to set the word length. For more information, see Construction Parameters for SystemC Types.

Port Direction Mapping

VHDL port directions are mapped to SystemC as follows:

VHDL	SystemC
in	sc_in<T>, sc_in_resolved, sc_in_rv<W>
out	sc_out<T>, sc_out_resolved, sc_out_rv<W>
inout	sc_inout<T>, sc_inout_resolved, sc_inout_rv<W>
buffer	sc_out<T>, sc_out_resolved, sc_out_rv<W>

VHDL to SystemC State Mapping

VHDL states are mapped to sc_logic, sc_bit, and bool as follows:

std_logic	sc_logic	sc_bit	bool
'U'	'X'	'0'	false
'X'	'X'	'0'	false
'0'	'0'	'0'	false
'1'	'1'	'1'	true
'Z'	'Z'	'0'	false
'W'	'X'	'0'	false
'L'	'0'	'0'	false

std_logic	sc_logic	sc_bit	bool
'H'	'1'	'1'	true
'-'	'X'	'0'	false

SystemC to VHDL State Mapping

SystemC type bool is mapped to VHDL boolean as follows:

bool	VHDL
false	false
true	true

SystemC type sc_bit is mapped to VHDL bit as follows:

sc_bit	VHDL
'0'	'0'
'1'	'1'

SystemC type sc_logic is mapped to VHDL std_logic states as follows:

sc_logic	std_logic
'0'	'0'
'1'	'1'
'Z'	'Z'
'X'	'X'

VHDL Instantiating Verilog

Once you have generated a component declaration for a Verilog module, you can instantiate the component just like any other VHDL component. You can reference a Verilog module in the entity aspect of a component configuration – all you need to do is specify a module name instead of an entity name. You can also specify an optional secondary name for an optimized sub-module. Further, you can reference a Verilog configuration in the configuration aspect of a VHDL component configuration - just specify a Verilog configuration name instead of a VHDL configuration name.

Verilog Instantiation Criteria Within VHDL

A Verilog design unit may be instantiated within VHDL if it meets the following criteria:

- The design unit is a module or configuration. UDPs are not allowed.
- The ports are named ports (see Modules with Unnamed Ports).
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in VHDL).

Component Declaration for VHDL Instantiating Verilog

A Verilog module that is compiled into a library can be referenced from a VHDL design as though the module is a VHDL entity. Likewise, a Verilog configuration can be referenced as though it were a VHDL configuration.

The interface to the module can be extracted from the library in the form of a component declaration by running vgencomp. Given a library and module name, vgencomp writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector
- vl_logic and vl_logic_vector

VHDL and Verilog Identifiers

The VHDL identifiers for the component name, port names, and generic names are the same as the Verilog identifiers for the module name, port names, and parameter names. Except for the cases noted below, ModelSim leaves the Verilog identifier alone when it generates the entity.

ModelSim converts Verilog identifiers to VHDL 1076-1993 extended identifiers in three cases:

- The Verilog identifier is not a valid VHDL 1076-1987 identifier.
- You compile the Verilog module with the **-93** argument. One exception is a valid, lowercase identifier (e.g., topmod). Valid, lowercase identifiers will not be converted even if you compile with **-93**.
- The Verilog identifier is not unique when case is ignored. For example, if you have TopMod and topmod in the same module, ModelSim will convert the former to \TopMod\).

vgencomp Component Declaration when VHDL Instantiates Verilog

vgencomp generates a component declaration according to these rules:

- Generic Clause

  A generic clause is generated if the module has parameters. A corresponding generic is defined for each parameter that has an initial value that does not depend on any other parameters.

  The generic type is determined by the parameter's initial value as follows:

Parameter value	**Generic type**
integer	integer
real	real
string literal	string

  The default value of the generic is the same as the parameter's initial value. For example:

Verilog parameter	**VHDL generic**
parameter p1 = 1 - 3;	p1 : integer := -2;
parameter p2 = 3.0;	p2 : real := 3.000000;
parameter p3 = "Hello";	p3 : string := "Hello";

- Port Clause

  A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named Verilog port.

  You can set the VHDL port type to bit, std_logic, or vl_logic. If the Verilog port has a range, then the VHDL port type is bit_vector, std_logic_vector, or vl_logic_vector. If the range does not depend on parameters, then the vector type will be constrained accordingly, otherwise it will be unconstrained. For example:

Verilog port	**VHDL port**
input p1;	p1 : in std_logic;
output [7:0] p2;	p2 : out std_logic_vector(7 downto 0);
output [4:7] p3;	p3 : out std_logic_vector(4 to 7);
inout [W-1:0] p4;	p4 : inout std_logic_vector;

  Configuration declarations are allowed to reference Verilog modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a Verilog instance to configure the instantiations within the Verilog module.

Modules with Unnamed Ports

Verilog allows modules to have unnamed ports, whereas VHDL requires that all ports have names. If any of the Verilog ports are unnamed, then all are considered to be unnamed, and it is not possible to create a matching VHDL component. In such cases, the module may not be instantiated from VHDL.

Unnamed ports occur when the module port list contains bit-selects, part-selects, or concatenations, as in the following example:

```
module m(a[3:0], b[1], b[0], {c,d});
  input [3:0] a;
  input [1:0] b;
  input c, d;
endmodule
```

Note that *a[3:0]* is considered to be unnamed even though it is a full part-select. A common mistake is to include the vector bounds in the port list, which has the undesired side effect of making the ports unnamed (which prevents the user from connecting by name even in an all-Verilog design).

Most modules having unnamed ports can be easily rewritten to explicitly name the ports, thus allowing the module to be instantiated from VHDL. Consider the following example:

```
module m(y[1], y[0], a[1], a[0]);
  output [1:0] y;
  input [1:0] a;
endmodule
```

Here is the same module rewritten with explicit port names added:

```
module m(.y1(y[1]), .y0(y[0]), .a1(a[1]), .a0(a[0]));
  output [1:0] y;
  input [1:0] a;
endmodule
```

"Empty" Ports

Verilog modules may have "empty" ports, which are also unnamed, but they are treated differently from other unnamed ports. If the only unnamed ports are "empty", then the other ports may still be connected to by name, as in the following example:

```
module m(a, , b);
  input a, b;
endmodule
```

Although this module has an empty port between ports "a" and "b", the named ports in the module can still be connected to from VHDL.

Verilog Instantiating VHDL

You can reference a VHDL entity or configuration from Verilog as though the design unit is a module or a configuration of the same name.

VHDL Instantiation Criteria Within Verilog

A VHDL design unit may be instantiated within Verilog if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type bit, bit_vector, std_ulogic, std_ulogic_vector, vl_ulogic, vl_ulogic_vector, or their subtypes. The port clause may have any mix of these types.
- The generics are of type integer, real, time, physical, enumeration, or string. String is the only composite type allowed.

Entity and Architecture Names and Escaped Identifiers

An entity name is not case sensitive in Verilog instantiations. The entity default architecture is selected from the work library unless specified otherwise. Since instantiation bindings are not determined at compile time in Verilog, you must instruct the simulator to search your libraries when loading the design. See Library Usage for more information.

Alternatively, you can employ the escaped identifier to provide an extended form of instantiation:

```
\mylib.entity(arch) u1 (a, b, c);
\mylib.entity u1 (a, b, c);
\entity(arch) u1 (a, b, c);
```

If the escaped identifier takes the form of one of the above and is not the name of a design unit in the work library, then the instantiation is broken down as follows:

- library = mylib
- design unit = entity
- architecture = arch

Named Port Associations

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

Named port associations are not case sensitive unless a VHDL port name is an extended identifier (1076-1993). If the VHDL port name is an extended identifier, the association is case sensitive and the VHDL identifier's leading and trailing backslashes are removed before comparison.

Generic Associations

Generic associations are provided via the module instance parameter value list. List the values in the same order that the generics appear in the entity. Parameter assignment to generics is not case sensitive.

The **defparam** statement is not allowed for setting generic values.

SDF Annotation

A mixed VHDL/Verilog design can also be annotated with SDF. See SDF for Mixed VHDL and Verilog Designs for more information.

SystemC Instantiating Verilog

To instantiate Verilog modules into a SystemC design, you must first create a SystemC Foreign Module (Verilog) Declaration for each Verilog module. Once you have created the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

Verilog Instantiation Criteria Within SystemC

A Verilog design unit may be instantiated within SystemC if it meets the following criteria:

- The design unit is a module (UDPs and Verilog primitives are not allowed).
- The ports are named ports (Verilog allows unnamed ports).
- The Verilog module name must be a valid C++ identifier.
- The ports are not connected to bidirectional pass switches (it is not possible to handle pass switches in SystemC).

A Verilog module that is compiled into a library can be instantiated in a SystemC design as though the module were a SystemC module by passing the Verilog module name to the foreign module constructor. For an illustration of this, see Example 9-1.

SystemC and Verilog Identifiers

The SystemC identifiers for the module name and port names are the same as the Verilog identifiers for the module name and port names. Verilog identifiers must be valid C++ identifiers. SystemC and Verilog are both case sensitive.

SystemC Foreign Module (Verilog) Declaration

In cases where you want to run a mixed simulation with SystemC and Verilog, you must generate and declare a foreign module that stands in for each Verilog module instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

After you have analyzed the design, you can generate a foreign module declaration with an scgenmod similar to the following:

```
scgenmod mod1
```

where *mod1* is a Verilog module. A foreign module declaration for the specified module is written to stdout.

Guidelines for Manual Creation in Verilog

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to Verilog ports. These ports must be explicitly named in the foreign module's constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module's HDL name to the sc_foreign_module base class constructor. For Verilog, the HDL name is simply the Verilog module name corresponding to the foreign module, or **[<lib>].<module>**.
- parameterized modules are allowed, see Parameter Support for SystemC Instantiating Verilog for details.

Example 9-1. SystemC Instantiating Verilog: Example 1

A sample Verilog module to be instantiated in a SystemC design is:

```
module vcounter (clock, topcount, count);
   input clock;
   input topcount;
   output count;
   reg count;
   ...
endmodule
```

The SystemC foreign module declaration for the above Verilog module is:

```
class counter : public sc_foreign_module {
   public:
   sc_in<bool> clock;
   sc_in<sc_logic> topcount;
   sc_out<sc_logic> count;
counter(sc_module_name nm)
   : sc_foreign_module(nm, "lib.vcounter"),
   clock("clock"),
   topcount("topcount"),
   count("count")
   {}
};
```

The Verilog module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the instance name of the Verilog module.

Example 9-2. SystemC Instantiating Verilog: Example 2

Another variation of the SystemC foreign module declaration for the same Verilog module might be:

```
class counter : public sc_foreign_module {
   public:
      ...
counter(sc_module_name nm, char* hdl_name)
   : sc_foreign_module(nm, hdl_name),
   clock("clock"),
      ...
{}
};
```

The instantiation of this module would be:

```
counter dut("dut", "lib.counter");
```

Parameter Support for SystemC Instantiating Verilog

Since the SystemC language has no concept of parameters, parameterized values must be passed from a SystemC parent to a Verilog child through the SystemC foreign module (sc_foreign_module). See SystemC Foreign Module (Verilog) Declaration for information regarding the creation of sc_foreign_module.

Generic Parameters in sc_foreign_module Constructor (Verilog)

To instantiate a Verilog module containing parameterized values into the SystemC design, you must pass two parameters to the sc_foreign_module constructor: the number of parameters (int num_generics), and the parameter list (const char* generic_list). The generic_list is listed as an array of const char*.

If you create your foreign module manually (see Guidelines for Manual Creation in Verilog), you must also pass the parameter information to the sc_foreign_module constructor. If you use **scgenmod** to create the foreign module declaration, the parameter information is detected in the HDL child and is incorporated automatically.

Example 9-3. SystemC Instantiating Verilog, Parameter Information

Following Example 9-1, let’s see the parameter information that would be passed to the SystemC foreign module declaration:

```
class counter : public sc_foreign_module {
   public:
      sc_in<bool> clk;
      ...
counter(sc_module_name nm, char* hdl_name
      int num_generics, const char** generic_list)
   : sc_foreign_module(nm, hdl_name, num_generics,
                     generic_list),
   {}
};
```

Example 9-4. SystemC Instantiating Verilog, Complete Example

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF
class ringbuf : public sc_foreign_module {
   public:
      sc_in<bool> clk;
      ...
counter(sc_module_name nm, char* hdl_name
      int num_generics, const char** generic_list)
   : sc_foreign_module(nm, hdl_name, num_generics,
                     generic_list),
   {}
};
#endif
-------------------------------------------------------------------------
// test_ringbuf.h
#ifndef INCLUDED_TEST_RINGBUF
#define INCLUDED_TEST_RINGBUF

#include "ringbuf.h"
#include "string.h"

SC_MODULE(test_ringbuf)
{
sc_signal<sc_logic> iclock;
...
   ...
// Verilog module instance
    ringbuf* chip;

    SC_CTOR(test_ringbuf)
      : iclock("iclock"),
      ...
      ...
   {
        const char* generic_list[3];

      generic_list[0] = strdup("int_param=4");
      generic_list[1] = strdup("real_param=2.6");
      generic_list[2] = strdup("str_param=\"Hello\"");
      // Enclose the string
      // in double quotes

      // Create Verilog module instance.
        chip = new ringbuf("chip", "ringbuf", 3, generic_list);
      // Connect ports
        chip->clock(iclock);
        ...
      // Cleanup the memory allocated for the generic list
      for (int i = 0; i < 3; i++;)
         free((char*)generic_list[i]);
      ...
    }

    ~test_ringbuf()
    {
        delete chip;
    }
};

#endif

-------------------------------------------------------------------------

// test_ringbuf.cpp
#include "test_ringbuf.h"

SC_MODULE_EXPORT(test_ringbuf);

--------------------------------------------------------------------

// ringbuf.v

`timescale 1ns / 1ns

module ringbuf (clock, reset, txda, rxda, txc, outstrobe);

    // Design Parameters Control Complete Design
    parameter int_param = 0;
    parameter real_param = 2.9;
    parameter str_param = "Error";

    // Define the I/O names
    input clock, txda, reset ;
    ...
    initial begin
        $display("int_param=%0d", int_param);
        $display("real_param=%g", real_param);
        $display("str_param=%s", str_param);
    end

endmodule
--------------------------------------------------------------------------
```

To run the simulation, use the following commands:

```
vsim1> vlib work
vsim2> vlog ringbuf.v
vsim3> scgenmod ringbuf > ringbuf.h
vsim4> sccom test_ringbuf.cpp
vsim5> sccom -link
vsim6> vsim -c test_ringbuf
```

The simulation returns:

```
# int_param=4
# real_param=2.6
# str_param=Hello
```

Verilog Instantiating SystemC

You can reference a SystemC module from Verilog as though the design unit is a module of the same name.

SystemC Instantiation Criteria for Verilog

A SystemC module can be instantiated in Verilog if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- SystemC modules are exported using the SC_MODULE_EXPORT macro. See Exporting SystemC Modules for Verilog.
- The module ports are as listed in the table shown in Channel and Port Type Mapping.
- Port data type mapping must match exactly. See the table in Data Type Mapping to Verilog.

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module declaration. Named port associations are case sensitive.

Exporting SystemC Modules for Verilog

To be able to instantiate a SystemC module from Verilog (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"
SC_MODULE_EXPORT(transceiver);
```

Parameter Support for Verilog Instantiating SystemC

Passing Parameters from Verilog to SystemC

To pass actual parameter values, simply use the native Verilog parameter override syntax. Parameters are passed to SystemC via the module instance parameter value list.

In addition to int, real, and string, ModelSim supports parameters with a bit range.

Named parameter association must be used for all Verilog modules that instantiate SystemC.

Retrieving Parameter Values

To retrieve parameter override information from Verilog, you can use the following functions:

```
void sc_get_param(const char* param_name, int& param_value);
void sc_get_param(const char* param_name, double& param_value);
void sc_get_param(const char* param_name, sc_string& param_value, char
format_char = 'a');
```

The first argument to sc_get_param defines the parameter name, the second defines the parameter value. For retrieving string values, ModelSim also provides a third optional argument, format_char. It is used to specify the format for displaying the retrieved string. The format can be ASCII ("a" or "A"), binary ("b" or "b"), decimal ("d" or "d"), octal ("o" or "O"), or hexadecimal ("h" or "H"). ASCII is the default.

Alternatively, you can use the following forms of the above functions in the constructor initializer list:

```
int sc_get_int_param(const char* param_name);
double sc_get_real_param(const char* param_name);
sc_string sc_get_string_param(const char* param_name, char format_char =
'a');
```

Example 9-5. Verilog Instantiating SystemC, Parameter Information

Here is a complete example, ring buffer, including all files necessary for simulation.

```
// test_ringbuf.v

`timescale 1ns / 1ps
module test_ringbuf();
    reg clock;
   ...
   parameter int_param = 4;
    parameter real_param = 2.6;
    parameter str_param = "Hello World";
    parameter [7:0] reg_param = 'b001100xz;

    // Instantiate SystemC module
    ringbuf #(.int_param(int_param),
              .real_param(real_param),
              .str_param(str_param),
              .reg_param(reg_param))
         chip(.clock(clock),
            ...
            ... };
   endmodule

-------------------------------------------------------------------------

// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF

#include <systemc.h>
#include "control.h"
...

SC_MODULE(ringbuf)
{
public:
    // Module ports
    sc_in clock;
    ...
   ...

    SC_CTOR(ringbuf)
        : clock("clock"),
      ...
      ...
{
   cout << "int_param="
      << sc_get_int_param("int_param") << endl;
   cout << "real_param="
      << sc_get_real_param("real_param") << endl;
   cout << "str_param="
      << (const char*)sc_get_string_param("str_param", 'a')
```

```
         << endl;
      cout << "reg_param="
         << (const char*)sc_get_string_param("reg_param", 'b')
         << endl;
       }

       ~ringbuf() {}
   };

   #endif

   ------------------------------------------------------------------------

   // ringbuf.cpp
   #include "ringbuf.h"

   SC_MODULE_EXPORT(ringbuf);
```

To run the simulation, you would enter the following commands:

```
vsim1> vlib work
vsim1> sccom ringbuf.cpp
vsim1> vlog test_ringbuf.v
vsim1> sccom -link
vsim1> vsim test_ringbuf
```

The simulation would return the following:

```
# int_param=4
# real_param=2.6
# str_param=Hello World
# reg_param=001100xz
```

SystemC Instantiating VHDL

To instantiate VHDL design units into a SystemC design, you must first generate a SystemC Foreign Module (Verilog) Declaration for each VHDL design unit you want to instantiate. Once you have generated the foreign module declaration, you can instantiate the foreign module just like any other SystemC module.

VHDL Instantiation Criteria Within SystemC

A VHDL design unit may be instantiated from SystemC if it meets the following criteria:

- The design unit is an entity/architecture pair or a configuration.
- The entity ports are of type bit, bit_vector, std_logic, std_logic_vector, std_ulogic, std_ulogic_vector, or their subtypes. The port clause may have any mix of these types. Only locally static subtypes are allowed. Also, array types must be locally static constrained array subtypes (e.g. std_logic_vector(8 downto 0), rather than std_logic_vector(4*2 downto 0), which is not supported.)

Port associations may be named or positional. Use the same port names and port positions that appear in the entity.

SystemC Foreign Module (VHDL) Declaration

In cases where you want to run a mixed simulation with SystemC and VHDL, you must create and declare a foreign module that stands in for each VHDL design unit instantiated under SystemC. The foreign modules can be created in one of two ways:

- running **scgenmod**, a utility that automatically generates your foreign module declaration (much like **vgencomp** generates a component declaration)
- modifying your SystemC source code manually

Only locally static subtypes are allowed with the use of **scgenmod**. Also, array types must be locally static constrained array subtypes (e.g. std_logic_vector(4*2 downto 0). A declaration such as std_logic_vector(generic_val downto 0), where generic_val is a generic, is not supported, since generics are not locally static.

After you have analyzed the design, you can generate a foreign module declaration with an scgenmod command similar to the following:

```
scgenmod mod1
```

Where *mod1* is a VHDL entity. A foreign module declaration for the specified entity is written to stdout.

Guidelines for Manual Creation in VHDL

Apply the following guidelines to the creation of foreign modules. A foreign module:

- contains ports corresponding to VHDL ports. These ports must be explicitly named in the foreign module’s constructor initializer list.
- must not contain any internal design elements such as child instances, primitive channels, or processes.
- must pass a secondary constructor argument denoting the module’s HDL name to the sc_foreign_module base class constructor. For VHDL, the HDL name can be in the format [<lib>.]<primary>[(<secondary>)] or [<lib>.]<conf>.
- generics are supported for VHDL instantiations in SystemC designs. See Generic Support for SystemC Instantiating VHDL for more information.

Example 9-6. SystemC Design Instantiating a VHDL Design Unit

A sample VHDL design unit to be instantiated in a SystemC design is:

```
entity counter is
   port (count : buffer bit_vector(8 downto 1);
      clk   : in bit;
      reset : in bit);
end;
architecture only of counter is
   ...
   ...

end only;
```

The SystemC foreign module declaration for the above VHDL module is:

```
class counter : public sc_foreign_module {
   public:
      sc_in<bool> clk;
      sc_in<bool> reset;
      sc_out<sc_logic> count;
counter(sc_module_name nm)
   : sc_foreign_module(nm, "work.counter(only)"),
   clk("clk"),
   reset("reset"),
   count("count")
   {}
};
```

The VHDL module is then instantiated in the SystemC source as follows:

```
counter dut("dut");
```

where the constructor argument (*dut*) is the VHDL instance name.

Generic Support for SystemC Instantiating VHDL

Since the SystemC language has no concept of generics, generic values must be passed from a SystemC parent to an HDL child through the SystemC foreign module (sc_foreign_module). See SystemC Foreign Module (Verilog) Declaration for information regarding the creation of sc_foreign_module.

Generic Parameters in sc_foreign_module Constructor (VHDL)

To instantiate a VHDL entity containing generics into the SystemC design, you must pass two parameters to the sc_foreign_module constructor: the number of generics (int num_generics), and the generic list (const char* generics_list). The generic_list is listed as an array of const char*.

If you create your foreign module manually (see Guidelines for Manual Creation in Verilog), you must also pass the generic information to the sc_foreign_module constructor. If you use **scgenmod** to create the foreign module declaration, the generic information is detected in the HDL child and is incorporated automatically.

Example 9-7. SystemC Instantiating VHDL, Generic Information

Following Example 9-6, let’s see the generic information that would be passed to the SystemC foreign module declaration. The generic parameters passed to the constructor are shown in magenta color:

```
class counter : public sc_foreign_module {
      public:

      sc_in<bool> clk;
      ...
counter(sc_module_name nm, char* hdl_name
      int num_generics, const char** generic_list)
   : sc_foreign_module(nm, hdl_name, num_generics,
                     generic_list),
   {}
};
```

The instantiation is:

```
dut = new counter ("dut", "work.counter", 9, generic_list);
```

Example 9-8. SystemC Instantiating VHDL, Using Generics

Here is another example, a ring buffer, complete with all files necessary for the simulation.

```
// ringbuf.h
#ifndef INCLUDED_RINGBUF
#define INCLUDED_RINGBUF
class ringbuf : public sc_foreign_module {
   public:
      sc_in<bool> clk;
      ...
counter(sc_module_name nm, char* hdl_name
      int num_generics, const char** generic_list)
   : sc_foreign_module(nm, hdl_name, num_generics,
                     generic_list),
   {}
};
#endif

-------------------------------------------------------------------------
// test_ringbuf.h
#ifndef INCLUDED_TEST_RINGBUF
#define INCLUDED_TEST_RINGBUF

#include "ringbuf.h"

SC_MODULE(test_ringbuf)
{
    sc_signal<T> iclock;
    ...
   ...
```

```
    // VHDL module instance
    ringbuf* chip;

    SC_CTOR(test_ringbuf)
      : iclock("iclock"),
      ...
      ...
   {
        const char* generic_list[9];

        generic_list[0] = strdup("int_param=4");
        generic_list[1] = strdup("real_param=2.6");
        generic_list[2] = strdup("str_param=\"Hello\"");
        generic_list[3] = strdup("bool_param=false");
        generic_list[4] = strdup("char_param=Y");
        generic_list[5] = strdup("bit_param=0");
        generic_list[6] = strdup("bv_param=010");
        generic_list[7] = strdup("logic_param=Z");
        generic_list[8] = strdup("lv_param=01XZ");

      // Cleanup the memory allocated for the generic list
      for (int = 0; i < 9; i++;)
         free((char*)generic_list[i]);
      // Create VHDL module instance.
        chip = new ringbuf("chip", "ringbuf", 9, generic_list);
};

#endif

-------------------------------------------------------------------------
-- test_ringbuf.cpp
#include "test_ringbuf.h"

SC_MODULE_EXPORT(test_ringbuf);

-------------------------------------------------------------------------
-- ringbuf.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE std.textio.all;

ENTITY ringbuf IS
    generic (
        int_param : integer;
        real_param : real;
        str_param : string;
        bool_param : boolean;
        char_param : character;
        bit_param : bit;
        bv_param : bit_vector(0 to 2);
        logic_param : std_logic;
        lv_param : std_logic_vector(3 downto 0));
    PORT (
        clock     : IN std_logic;
      ..
      ...
   );
```

```
END ringbuf;


ARCHITECTURE RTL OF ringbuf IS

BEGIN
    print_param: PROCESS
        variable line_out: Line;
    BEGIN
        write(line_out, string'("int_param="), left);
        write(line_out, int_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("real_param="), left);
        write(line_out, real_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("str_param="), left);
        write(line_out, str_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("bool_param="), left);
        write(line_out, bool_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("char_param="), left);
        write(line_out, char_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("bit_param="), left);
        write(line_out, bit_param);
        writeline(OUTPUT, line_out);
        write(line_out, string'("bv_param="), left);
        write(line_out, bv_param);
        writeline(OUTPUT, line_out);
        WAIT FOR 20 NS;
    END PROCESS;
END RTL;
```

To run the parameterized design, use the following commands.

```
vsim1> vlib work
vsim2> vcom ringbuf.vhd
vsim3> scgenmod ringbuf > ringbuf.h //creates the sc_foreign_module
                                            including generic mapping info
vsim4> sccom test_ringbuf.cpp
vsim5> sccom -link
vsim6> vsim -c test_ringbuf
```

The simulation returns the following:

```
# int_param=4
# real_param=2.600000e+00
# str_param=Hello
# bool_param=FALSE
# char_param=Y
# bit_param=0
# bv_param=010
```

VHDL Instantiating SystemC

To instantiate SystemC in a VHDL design, you must create a component declaration for the SystemC module. Once you have generated the component declaration, you can instantiate the SystemC component just like any other VHDL component.

SystemC Instantiation Criteria for VHDL

A SystemC design unit may be instantiated within VHDL if it meets the following criteria:

- SystemC module names are case sensitive. The module name at the SystemC instantiation site must match exactly with the actual SystemC module name.
- The SystemC design unit is exported using the SC_MODULE_EXPORT macro.
- The module ports are as listed in the table in Data Type Mapping to VHDL
- Port data type mapping must match exactly. See the table in Port Type Mapping.

Port associations may be named or positional. Use the same port names and port positions that appear in the SystemC module. Named port associations are case sensitive.

Component Declaration for VHDL Instantiating SystemC

A SystemC design unit can be referenced from a VHDL design as though it is a VHDL entity. The interface to the design unit can be extracted from the library in the form of a component declaration by running **vgencomp**. Given a library and a SystemC module name, **vgencomp** writes a component declaration to standard output.

The default component port types are:

- std_logic
- std_logic_vector

Optionally, you can choose:

- bit and bit_vector

VHDL and SystemC Identifiers

The VHDL identifiers for the component name and port names are the same as the SystemC identifiers for the module name and port names. If a SystemC identifier is not a valid VHDL 1076-1987 identifier, it is converted to a VHDL 1076-1993 extended identifier (in which case you must compile the VHDL with the **-93** or later switch).

vgencomp Component Declaration when VHDL Instantiates SystemC

vgencomp generates a component declaration according to these rules:

- Port Clause

  A port clause is generated if the module has ports. A corresponding VHDL port is defined for each named SystemC port.

  You can set the VHDL port type to bit or std_logic. If the SystemC port has a range, then the VHDL port type is bit_vector or std_logic_vector. For example:

SystemC port	**VHDL port**
sc_in<sc_logic>p1;	p1 : in std_logic;
sc_out<sc_lv<8>>p2;	p2 : out std_logic_vector(7 downto 0);
sc_inout<sc_lv<8>>p3;	p3 : inout std_logic_vector(7 downto 0)

  Configuration declarations are allowed to reference SystemC modules in the entity aspects of component configurations. However, the configuration declaration cannot extend into a SystemC instance to configure the instantiations within the SystemC module.

Exporting SystemC Modules for VHDL

To be able to instantiate a SystemC module within VHDL (or use a SystemC module as a top level module), the module must be exported.

Assume a SystemC module named *transceiver* exists, and that it is declared in header file *transceiver.h*. Then the module is exported by placing the following code in a *.cpp* file:

```
#include "transceiver.h"
SC_MODULE_EXPORT(transceiver);
```

The **sccom -link** command collects the object files created in the work library, and uses them to build a shared library (.so) in the current work library. If you have changed your SystemC source code and recompiled it using **sccom**, then you must run **sccom -link** before invoking **vsim**. Otherwise your changes to the code are not recognized by the simulator.

Generic Support for VHDL Instantiating SystemC

Support for generics is available in a workaround flow for the current release. For workaround flow details, please refer to *systemc_generics.note* located in the *<install_dir>/modeltech/docs/technotes* directory.

Chapter 10 Transactions

Introduction

A transaction is a statement of what the design is doing between one time and another during a simulation run. It is just that simple.

Transactions vs. Transaction Level Modeling (TLM)

While the definition of a transaction may be simple, the word "transaction" itself can be confusing because of its association with Transaction Level Modeling. In TLM, design units pass messages across interfaces and these messages are typically called transactions. We use the term "transaction" in a broader sense: it is an abstract statement, logged in the WLF file, of what the design was doing at a specific time.

The transaction is included in the designer’s source code and logged into the WLF file. Often, transactions represent packets of data moving around between design objects. Transactions allow users to debug and monitor the design at any level of abstraction.

You can use the SystemC Verification (SCV) library to record transactions for viewing with the ModelSim tool. You create and record transactions through published API calls in your design source code. As simulation progresses, individual transactions are recorded into the WLF file and are available for design debug and performance analysis in both interactive debug and post-simulation debug.

Transaction Definitions

Minimally, a *transaction* consists of a name, a start time, and an end time. With that alone, you could record the transitions of a state machine, summarize the activity on a bus, and so forth. Transactions can also be assigned user-defined *attributes*, such as address, data, status, and so on.

Much as values are recorded on wires and signals, transactions are recorded on *streams*. Streams are debuggable objects: they appear in or may be added to GUI windows such as the Objects pane or Wave windows. The tool creates *substreams* as needed so that overlapping transactions on the stream remain distinct. Overlapping transactions can be drawn either as *parallel*, where no specific relationship exists between the two transactions; or *phase*, where the overlapping transaction is actually a “child” of the initial transaction.

Viewing Transactions

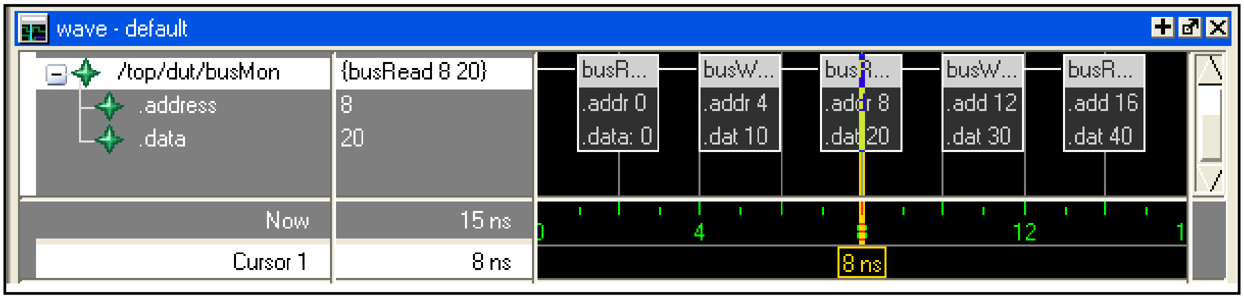
Viewing transactions in the GUI is intended to be as intuitive as possible. Thus, most of the usual mouse-based operations, such as drag-and-drop, function similarly for transaction streams, substreams, attributes and elements of attributes.

Wave Window

Transaction stream objects are denoted by a four pointed star (green for SystemC). Streams which have objects below them appear with a plus icon to indicate that they can be expanded to reveal substreams and any attributes.

A Simple Transaction

Transactions are best viewed in the wave window. The figure below shows a transaction stream from a bus monitor. The transactions include simple, user-defined address and data attributes:



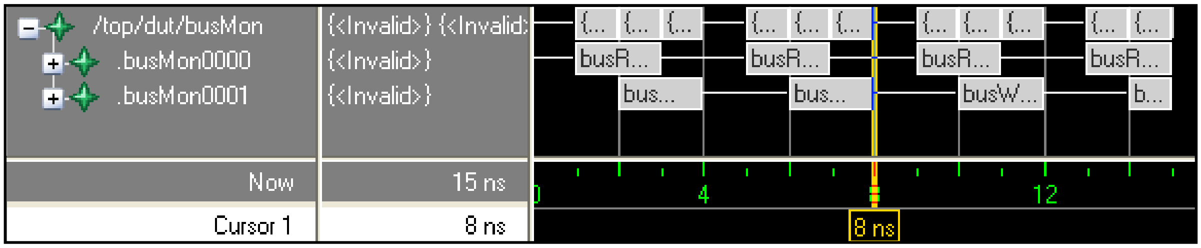
wave - default

/top/dut/busMon {busRead 8 20}
.address 8
.data 20
Now 15 ns
Cursor 1 8 ns

A transaction in the wave window appears as a box surrounding the transaction elements. The left edge of the box indicates the start time for the transaction, the right edge indicates the end time. A transaction that occurs in zero time appears as a simple vertical line. The top row of a transaction is the "Tag" or the kind of transaction. When the transaction stream is expanded, as it is in the picture above, additional rows are revealed that represent attributes of the transaction.

Overlapping Transactions

When more than one transaction is recorded on a stream at one time, the transactions are overlapping. The tool creates a separate substream for each transaction so that they are distinct from each other in the view. The stream expands to reveal the different substreams in this case. Expanding the substream reveals attributes on those transactions. They appear in the wave window as follows:



Substream Creation

Substreams are created so as to make debug possible. Without them, overlapping transactions would obscure each other. That said, you have no direct control over the creation of substreams; the tool creates them as needed. It uses a very simple rule: a transaction is placed on the first substream that has no active transaction and does not have any transaction in the future of the one being logged.

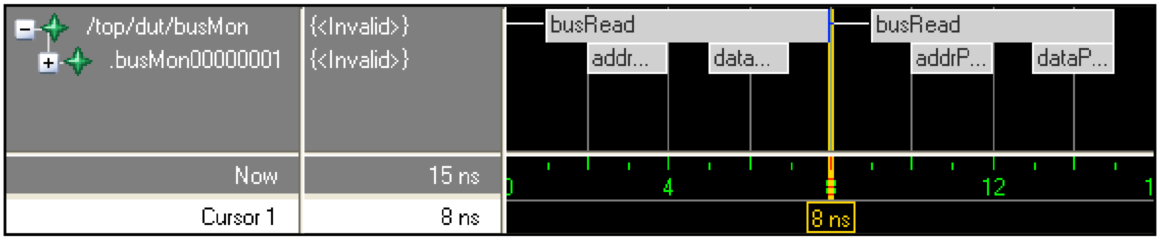
Retroactive Recording

Retroactive recording refers to the recording of a transaction that occurred in the past. When drawing transactions, this can lead to more substreams being added than you might expect. For example, you might have a substream with a “blank” period between 10 ns and 20 ns, during which no transactions are recorded, and you want to record a transaction retroactively between 11 and 16 ns. Even though there might be a “space” between the two transactions long enough for your retroactive transaction, the tool creates an additional substream to draw that transaction. It does this because the cost of tracking all available "spaces" in the past is too high; the simpler solution is to track the end of the last transaction on each substream only.

You can control the number retroactive recording channels that are allowed in the WLF file by setting the RetroChannelLimit in the *modelsim.ini* file. Setting the variable to 0 turns off retroactive recording. Setting the limit too high can risk your performance, and the WLF file may not operate.

Drawing Phase / Child Transactions

The tool has an alternative way of drawing transactions that are considered to be “phases” of a “parent” transaction. Thus, you can consider them as “children” of the initial transaction. For example, consider that a busRead transaction may have several steps or phases. Each of these could be represented as a smaller, overlapping transaction and would appear on a second substream. However, you can indicate that these are phase, and by doing so, you instruct the tool to draw them specially, as shown in the following figure:



See Specifying and Recording Phase Transactions for information on how to specify the recording of a transaction as a phase of a parent transaction.

Structure Pane

Transactions are events occurring on transaction objects, otherwise known as streams. Stream objects look like signals in the design hierarchy. When you navigate to a region containing a stream, the stream is visible in the Objects pane or in the Transcript area in response to a **show** command.

Objects Pane

Streams appear as simple or composite signals, depending on the complexity of transactions that have been defined for the stream. The icon for a stream is a four-point star in the color of the source language for the region in which the stream is found. Thus, a stream in a SystemC instance has a green star as its icon.

If the stream is composite (has attributes or phase/child transactions), there is an expand button to the left of the icon as with any composite object. A substream has an expand button if it in turn has a substream or if transactions on that substream have user-specified attributes. An attribute has an expand button if its value is a composite value, such as an array or structure.

If there have been no transactions defined on the stream, or none of the transactions have attributes, then the stream is a simple signal with the tag of the current transaction as its value. When no transaction is active, the value is "<Inactive>". Otherwise, the value is the tag of the active transaction, such as "busRetry".

List Window

Transaction streams, substreams, attributes and attribute elements may be added to the list window alongside HDL items.

For HDL items, the List window normally prints a row containing the values of all selected design elements at each time or delta advance. This makes sense, as one expects that they have only one value per time step and the deltas help to show the underlying simulation behavior, interaction of processes, etc. Transactions, however, may have significant events in the same delta: one transaction may end and another may begin without any time or delta advance.

For transactions, the List window prints a row any time a transaction’s state changes. Specifically, rows are printed when a transaction starts or ends, and when any attribute changes state.

The following is an example of a list output for a stream, showing a begin attribute, a special attribute and an end attribute in the style of SCV:

```
ns /top/abc/busMon
   delta
0  +0 <Inactive>
1  +0 <Inactive>
1  +0 <Inactive>
1  +0 {busRead 1 <Inactive> <Inactive>}
3  +0 {busRead 1 100 <Inactive>}
3  +0 {busRead 1 100 10}
3  +0 <Inactive>
4  +0 <Inactive>
4  +0 <Inactive>
4  +0 <Inactive>
...
```

In the list above, the same time/delta repeats itself as changes are made to the transaction. For example, at 1(0) a busRead begins with the begin attribute set to the value “1”. At time 3(0), the end attribute value “100” arrives, and so on.

Recording Transactions with SCV

SystemC users use the SCV library’s transaction recording API routines to define transactions, to start them, to end them, to create relationships between them and to attach additional information (attributes) to them. These routines are described in the SystemC Verification Standard Specification, Version 1.0e. Please refer to this documentation for SCV specific details.

This section highlights specific steps you must take to record transactions in a WLF file. SCV recording in ModelSim requires only one addition to your source code: the design must call **scv_tr_wlf_init()** to log transactions in the WLF file (see Initializing the Library and Creating the WLF Database).

Note

You must include the SCV libraries as part of the build process using the **-scv** argument to the tool’s SystemC compilation command sccom.

Steps for Recording in SCV

The steps can be summarized as follows:

- Initialize SCV and the MTI extensions for transaction recording and debug.
- Create a database tied to WLF.
- Provide SCV extensions for user-defined types used with attributes.
- Define transaction streams.
- Define transaction kinds.
- Start transactions.
- Record special attributes.
- End transactions.

Recording of attributes and relationships can occur at different points between (and including) the start and end of the transaction(s) as specified in the SCV API.

Additional actions can include:

- Specify relationships between transactions.
- Specify the begin and/or end time for a transaction.
- Control database logging.

The following sections documents any areas where the ModelSim tool's implementation deviates from the SCV specification.

Initializing the Library and Creating the WLF Database

The design must initialize the SCV library once as part of its own initialization. In your code, you issue a call to **scv_startup()**, followed by **scv_tr_wlf_init()**. After this, transactions are written to specific database objects in the code. You can create many objects, or create one and specify it as the default object.

The following is an example of a one-time initialization routine that sets up SCV, ties all databases to WLF and then creates one database as the default:

```
static scv_tr_db * init_recording() {
   scv_tr_db *txdb;

   /* Initialize SCV: */
   scv_startup();

   /* Tie databases to WLF: */
   scv_tr_mti_init();
```

```
      /* Create the new DB and make it the default: */
      txdb = new scv_tr_db("txdb");

      if (txdb != NULL)
         scv_tr_db::set_default_db(txdb);

   return txdb;
```

The ModelSim tool ignores the name argument to **scv_tr_db()** since all databases are tied to the WLF file once the user calls **scv_tr_wlf_init()**.

The ModelSim tool also ignores the **sc_time_unit** argument to **scv_tr_db()** when the database is a WLF database. The time unit of the database is specified by the overall simulation time unit.

Preparing Attribute Types

If your design uses standard C and SC types for attributes, no preparation work is needed with SCV, since SCV Extensions are defined for those types. For classes, structures or other user-defined types, you must provide SCV extensions so that SCV and the ModelSim tool can extract the necessary type and composition information to record the type. An example is as follows:

```
struct busAddrAttr {
   unsigned _addr;
};

template<>
class scv_extensions<busAddrAttr> : public
scv_extensions_base<busAddrAttr>
{
   public:
   scv_extensions<unsigned> _addr;
   SCV_EXTENSIONS_CTOR(busAddrAttr) { SCV_FIELD(_addr); }
};
```

All C/C++ SystemC types are supported, with the exception of those listed in the section, Unsupported Types.

Naming Attributes

Legal C-language identifiers are recommended for naming attributes, as these are guaranteed to be supported for debug.

Note

The ModelSim tool issues a warning for a non-standard name.

Defining Transaction Streams

Transactions are written on streams just the way ones and zeros are driven onto wires in a design. Therefore, the design must create one or more stream objects. Streams are tied to a specific database so that all transactions on them are written into that database only. Usually, the code declares the stream as a member of the module that will use it. Then, it must call the constructor, passing the stream's name (see Naming Streams) and database as parameters:

```
SC_MODULE(busModel)
{
   public:
      scv_tr_db *txdb;
      scv_tr_stream busStream;

      SC_CTOR(busModel) :
         txdb(init_recording()),
         busStream( "busModel", "**TRANSACTOR**")
   {
   }
}
```

This example code declares the database and stream objects. In the module constructor, it initializes the database by calling the setup routine. It initializes the stream object with its display name and a string indicating the stream kind. The database is presumed to be the default, though we could have been explicit and passed "txdb" as a third parameter.

Naming Transactions

All transactions must have names; anonymous transactions are not allowed. There are no restrictions on the names, though we recommend using standard C-language identifiers.

There are two ways to name a transaction. You can either specify the name to the generator, or specify it as an attribute to the transaction.

Naming Streams

You must provide a name for streams so that they can be referenced for debug. Anonymous streams are not allowed. Any name can be used, however, only legal C-language identifiers are recommended since these are guaranteed to be supported for debug.

Note

The ModelSim tool issues a warning for a non-standard name.

Substream Names

The tool names substreams automatically. The name of any substream is the first character of the parent’s name followed by a simple index number. The first substream has the index zero. If the parent stream has a non-standard name, such as one that starts with a numeral or a space, you may have difficulty with debug.

Defining Transaction Kinds

In SCV, each transaction is defined by a generator object, a kind of template for a transaction. Each object specifies the tag of the transaction and optional begin and end attributes. That is, begin and end attributes are part of the generator for that kind and are treated as part of each instance of that transaction.

First, the code must declare each generator, usually in the module in which it is to be used. Any begin and end attribute types must be provided as template parameters. Then, the generator must be constructed, usually in the constructor initialization list of the parent module:

```
SC_MODULE(busModel)
{
   public:
      scv_tr_db *txdb;
      scv_tr_stream busStream;
      scv_tr_generator<busAddrAttr, busDataAttr> busRead;

      SC_CTOR(busModel) :
         txdb(init_recording()),
         busStream( "busModel", "**TRANSACTOR**"),
         busRead("busRead", busStream)
      {
      }
}
```

The third and fourth arguments to **scv_tr_generator::scv_tr_generator()** are the names for the begin and end attributes. SCV allows these to be NULL by default.

The above example adds the declaration of the generator "busRead" and shows how the constructor is provided with a name (also "busRead") and the stream on which it will be used. The begin and end attributes are left anonymous.

Note

The ModelSim tool allows the design to re-use an attribute name with a different type. However, a warning is issued since this practice limits debug in the tool.

Starting Transactions

To issue a transaction, the design sets the value for the begin attribute and calls **scv_tr_generator::begin_transaction(...)** with the appropriate parameters as defined in the SCV API:

```
scv_tr_handle txh;
busAddrAttr busAddr;

busAddr._addr = 0x00FC01;

txh = busRead.begin_transaction(busAddr);
```

In this example, only the begin attribute "busAddr" is passed to **::begin_transaction()**. Other parameters may be used to specify relationships or specify a begin time other than the current simulation time (see Specifying Start and End Times).

Your design is not required to specify the values of begin attributes, even if they are part of the generator. For example, if your design had done this:

```
txh = busRead.begin_transaction();
```

... the value of the begin attribute would be considered undefined. The ModelSim tool allows this and indicates that such values are undefined for a specific transaction instance.

Recording Special Attributes

Special attributes are not part of the original transaction generator: they are afterthoughts. The type is defined as discussed in the section entitled Preparing Attribute Types, but they are recorded on the specific transaction instance through the transaction handle:

```
if (status != BUS_OK) {
   errorAttr err;

   err.code = status;

   txh.record_attribute(err);
}
```

Multiple Uses of Same Special Attribute

There is nothing to prevent your design from setting the same special attribute many times during the transaction. However, the ModelSim tool records only the last value of the attribute prior to the end of the transaction.

Once any attribute is used, special or otherwise, it is considered an attribute of the parent stream from that time onward. Thus, it shows up as a parameter on all subsequent transactions, even if it is unused.

Ending Transactions

You can end transactions in your code by placing a call to **scv_tr_generator::end_transaction()**, specifying any end attributes or other parameters:

```
busDataAttr busData;
```

```
busData._data = 10;
busRead.end_transaction(txh, busData);
```

In this example, only the transaction handle "txh" and the end attribute "busData" is passed to **::end_transaction()**. Other parameters may be used to specify relationships or specify an end time other than the current simulation time. See Specifying Start and End Times for details.

The design is not required to specify the values of end attributes, even if they are part of the generator. For example, if the design had done this:

```
busRead.end_transaction(txh);
```

... the value of the end attribute would be considered undefined. The ModelSim tool allows this and indicates that such values are undefined for a specific transaction instance.

Specifying and Recording Phase Transactions

Phase (child) transactions are unique to ModelSim. If recorded, they appear as transactions attached to their parent. The SCV specification does not describe this kind of transaction, but ModelSim can record it.

To record phase transactions, you must specify the “phase” relation and provide an appropriate parent transaction handle in a call to **::begin_transaction()**. Any transaction may have phases, including another phase transaction.

You can specify your own relation name for phases by modifying the value of the variable ScvPhaseRelationName in the *modelsim.ini* from “phase” to something else, such as “child”. This variable applies to recording only; once a phase is recorded in a WLF file, it is drawn as a phase, regardless of the setting of this variable.

Specifying Start and End Times

The SCV API allows you to specify a start and/or end time for any transaction. These are passed as parameters to **::begin_transaction()** and **::end_transaction()**. The time must be the current simulation time or earlier.

Start and End Times for Phase Transactions

The start and end times for phase (child) transactions must be entirely within the range of the parent transaction. In other words, the start time for the phase transaction must match or be later than the parent transaction’s start time, and the end time must match or be earlier than the parent’s end time.

Enabling and Disabling Logging

For transactions, all logging control is through SCV API routines. By default, when your design creates a stream, logging is enabled for that stream. You can enable and disable logging for the entire TR database through calls to **scv_tr_db::set_recording()**. There is no way to disable recording on a single stream. Further, there is no way in the ModelSim tool to distinguish a stream that is disabled from one that is merely inactive.

Limitations on SCV

Unsupported Types

All C/C++ aggregate types and atomic types are supported (array, class, struct, union, sc_bit, sc_logic), with the exception of the following types.

Unsupported C++ Atomic Types

- bit-field
- T* (pointer)

Unsupported C/C++ Aggregate Types

C/C++ aggregate types are not supported.

Unsupported SC Fixed-Point Types

Most C/C++ and SystemC types are supported, however the following are limited or are not supported at this time:

- sc_fix, sc_fix_fast
- sc_fixed, sc_fixed_fast
- sc_ufix, sc_ufix_fast
- sc_ufixed, sc_ufixed_fast

Recording in one WLF

You can record transactions in one WLF file at a time. The SCV API routines allow you to create and use multiple databases. However, if the chosen database is WLF, all databases are aliased to the same WLF file. Once created, you may load multiple WLF files that contain transactions into ModelSim for viewing and debugging.

Relation Recording

Relations are not recorded in the WLF database in the current release.

Chapter 11
WLF Files (Datasets) and Virtuals

This chapter describes the Wave Log Format (WLF) file and how you should and can use it in your simulation flow.

A ModelSim simulation can be saved to a wave log format (WLF) file for future viewing or comparison to a current simulation. We use the term "dataset" to refer to a WLF file that has been reopened for viewing.

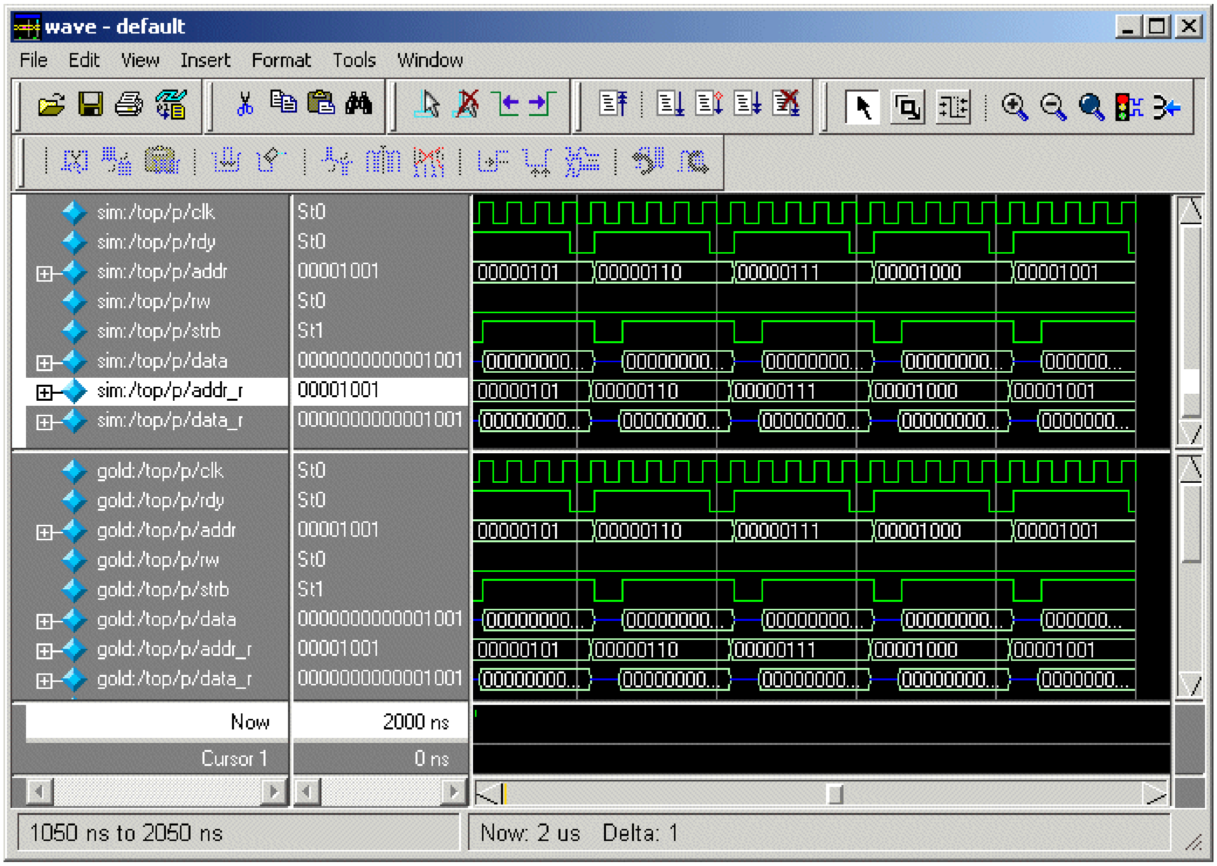
You can open more than one WLF file for simultaneous viewing. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different datasets.

WLF files are recordings of simulation runs. The WLF file is written as an archive file in binary format and is used to drive the debug windows at a later time. The files contain data from logged objects (e.g., signals and variables) and the design hierarchy in which the logged objects are found. You can record the entire design or choose specific objects.

The WLF file provides you with precise in-simulation and post-simulation debugging capability. Any number of WLF files can be reloaded for viewing or comparing to the active simulation.

A dataset is a previously recorded simulation that has been loaded into ModelSim. Each dataset has a logical name to let you indicate the dataset to which any command applies. This logical name is displayed as a prefix. The current, active simulation is prefixed by "sim:", while any other datasets are prefixed by the name of the WLF file by default.

Two datasets are displayed in the Wave window below. The current simulation is shown in the top pane and is indicated by the "sim" prefix. A dataset from a previous simulation is shown in the bottom pane and is indicated by the "gold" prefix.



sim:/top/p/clk	St0
sim:/top/p/rdy	St0
sim:/top/p/addr	00001001
sim:/top/p/rw	St0
sim:/top/p/strb	St1
sim:/top/p/data	0000000000001001
sim:/top/p/addr_r	00001001
sim:/top/p/data_r	0000000000001001
gold:/top/p/clk	St0
gold:/top/p/rdy	St0
gold:/top/p/addr	00001001
gold:/top/p/rw	St0
gold:/top/p/strb	St1
gold:/top/p/data	0000000000001001
gold:/top/p/addr_r	00001001
gold:/top/p/data_r	0000000000001001
Now	2000 ns
Cursor 1	0 ns

1050 ns to 2050 ns

Now: 2 us Delta: 1

The simulator resolution (see Simulator Resolution Limit (Verilog) or Simulator Resolution Limit (VHDL)) must be the same for all datasets you are comparing, including the current simulation. If you have a WLF file that is in a different resolution, you can use the wlfman command to change it.

Saving a Simulation to a WLF File

If you add objects to the Dataflow, List, or Wave windows, or log objects with the **log** command, the results of each simulation run are automatically saved to a WLF file called *vsim.wlf* in the current directory. If you run a new simulation in the same directory, the *vsim.wlf* file is overwritten with the new results.

If you want to save the WLF file and not have it be overwritten, select the dataset tab in the Workspace and then select **File** > **Save**. Or, you can use the **-wlf <filename>** argument to the vsim command or the dataset save command.

Note

If you do not use **dataset save** or **dataset snapshot**, you must end a simulation session with a **quit** or **quit -sim** command in order to produce a valid WLF file. If you don't end the simulation in this manner, the WLF file will not close properly, and ModelSim may issue the error message "bad magic number" when you try to open an incomplete dataset in subsequent sessions. If you end up with a "damaged" WLF file, you can try to "repair" it using the wlfrecover command.

WLF File Parameter Overview

There are a number of WLF file parameters that you can control via the *modelsim.ini* file or a simulator argument. This section summarizes the various parameters.

Table 11-1.

Feature	vsim argument	modelsim.ini	Default
WLF Filename	-wlf <filename>	WLFFilename=<filename>	*vsim.wlf*
WLF Size Limit	-wlfslim <n>	WLFSizeLimit = <n>	no limit
WLF Time Limit	-wlftlim <t>	WLFTimeLimit = <t>	no limit
WLF Compression	-wlfcompress -wlfnocompress	WLFCompress = 0|1	1 (-wlfcompress)
WLF Optimization[1]	-wlfopt -wlfnoopt	WLFOptimize = 0|1	1 (-wlfopt)
WLF Delete on Quit[a]	-wlfdeleteonquit -wlfnodeleteonquit	WLFDeleteOnQuit = 0|1	0
WLF Cache Size[a]	-wlfcachesize <n>	WLFCacheSize = <n>	256
WLF Collapse Mode	-wlfnocollapse -wlfcollapsedelta -wlfcollapsetime	WLFCollapseModel = 0|1|2	1

[1] These parameters can also be set using the dataset config command.

- WLF Filename — Specify the name of the WLF file.
- WLF Size Limit — Limit the size of a WLF file to <n> megabytes by truncating from the front of the file as necessary.
- WLF Time Limit — Limit the size of a WLF file to <t> time by truncating from the front of the file as necessary.
- WLF Compression — Compress the data in the WLF file.
- WLF Optimization — Write additional data to the WLF file to improve draw performance at large zoom ranges. Optimization results in approximately 15% larger

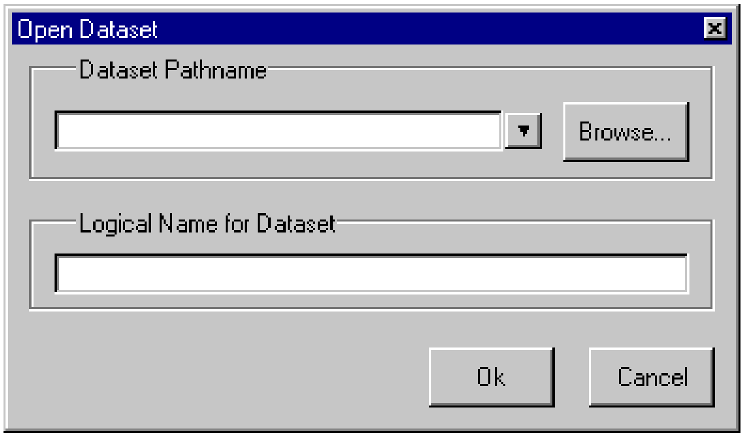
WLF files. Disabling WLF optimization also prevents ModelSim from reading a previously generated WLF file that contains optimized data.

- WLF Delete on Quit — Delete the WLF file automatically when the simulation exits. Valid for current simulation dataset (*vsim.wlf*) only.
- WLF Cache Size — Specify the size in megabytes of the WLF reader cache. WLF reader cache is enabled by default. The default value is 256. This feature caches blocks of the WLF file to reduce redundant file I/O. If the cache is made smaller or disabled, least recently used data will be freed to reduce the cache to the specified size.
- WLF Collapse Mode —WLF event collapsing has three settings: disabled, delta, time:
    - When disabled, all events and event order are preserved.
    - Delta mode records an object's value at the end of a simulation delta (iteration) only. Default.
    - Time mode records an object's value at the end of a simulation time step only.

Opening Datasets

To open a dataset, do one of the following:

- Select **File** > **Open** and choose Log Files or use the dataset open command.



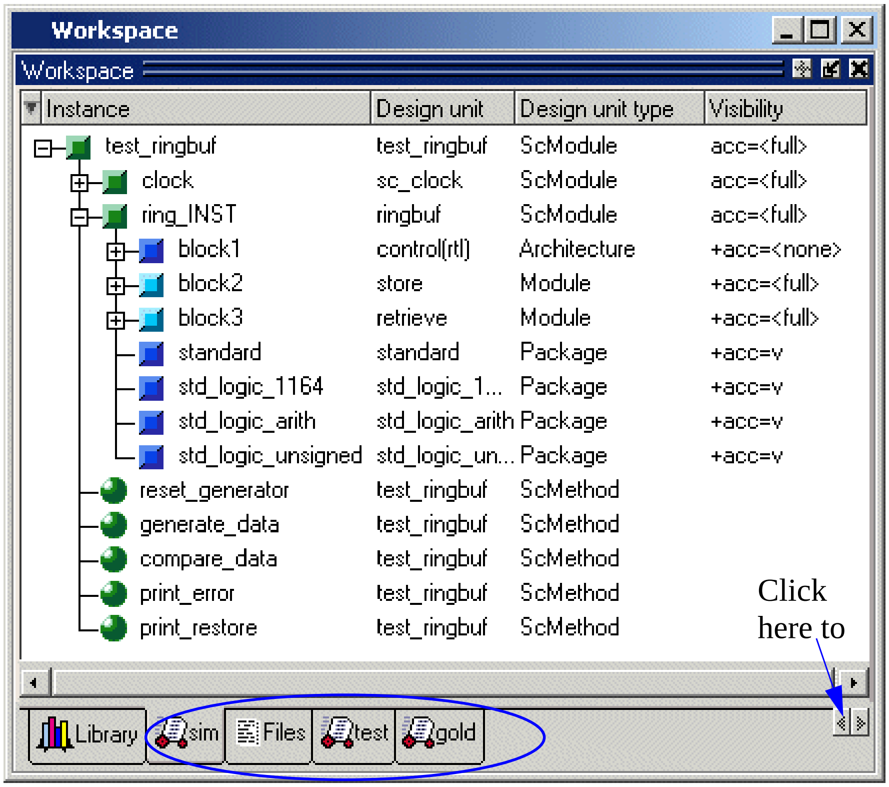
The Open Dataset dialog includes the following options:

- **Dataset Pathname** — Identifies the path and filename of the WLF file you want to open.
- **Logical Name for Dataset** — This is the name by which the dataset will be referred. By default this is the name of the WLF file.

Viewing Dataset Structure

Each dataset you open creates a structure tab in the Main window workspace. The tab is labeled with the name of the dataset and displays a hierarchy of the design units in that dataset.

The graphic below shows three structure tabs: one for the active simulation (*sim*) and one each for two datasets (*test* and *gold*).



Instance	Design unit	Design unit type	Visibility
test_ringbuf	test_ringbuf	ScModule	acc=<full>
clock	sc_clock	ScModule	acc=<full>
ring_INST	ringbuf	ScModule	acc=<full>
block1	control(rtl)	Architecture	+acc=<none>
block2	store	Module	+acc=<full>
block3	retrieve	Module	+acc=<full>
standard	standard	Package	+acc=v
std_logic_1164	std_logic_1...	Package	+acc=v
std_logic_arith	std_logic_arith	Package	+acc=v
std_logic_unsigned	std_logic_un...	Package	+acc=v
reset_generator	test_ringbuf	ScMethod	
generate_data	test_ringbuf	ScMethod	
compare_data	test_ringbuf	ScMethod	
print_error	test_ringbuf	ScMethod	
print_restore	test_ringbuf	ScMethod	

Click here to

If you have too many tabs to display in the available space, you can scroll the tabs left or right by clicking the arrow icons at the bottom right-hand corner of the window.

Structure Tab Columns

Each structure tab displays four columns by default:

Table 11-2.

Column name	Description
Instance	the name of the instance
Design unit	the name of the design unit
Design unit type	the type (e.g., Module, Entity, etc.) of the design unit

Table 11-2.

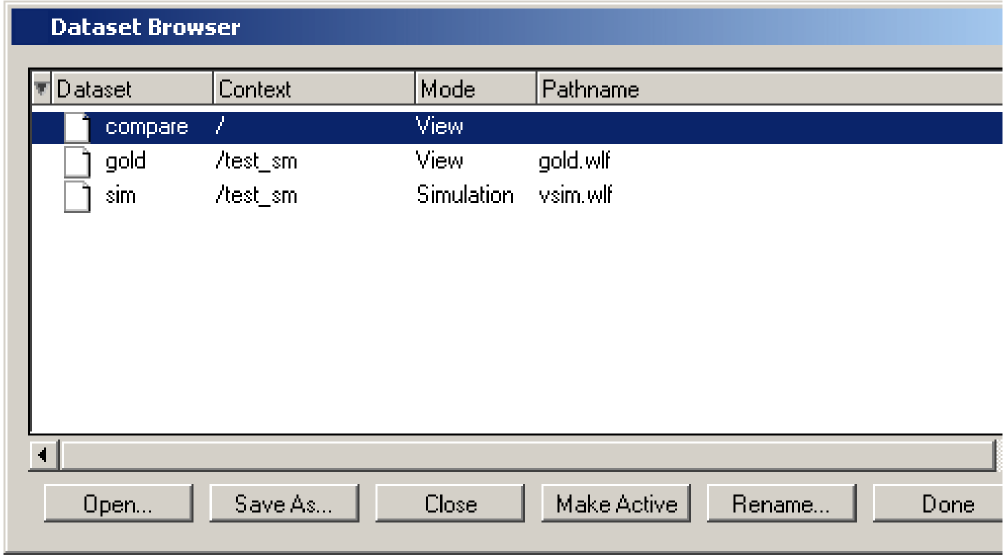
Column name	Description
Visibility	the current visibility of the object as it relates to design optimization; see Design Object Visibility and the +acc Argument for more information

Aside from the four columns listed above, there are numerous columns related to code coverage that can be displayed in structure tabs. You can hide or show columns by right-clicking a column name and selecting the name on the list.

Managing Multiple Datasets

GUI

When you have one or more datasets open, you can manage them using the **Dataset Browser**. To open the browser, select **View** > **Datasets**.



Dataset	Context	Mode	Pathname
compare	/	View	
gold	/test_sm	View	gold.wlf
sim	/test_sm	Simulation	vsim.wlf

Command Line

You can open multiple datasets when the simulator is invoked by specifying more than one **vsim -view <filename>** option. By default the dataset prefix will be the filename of the WLF file. You can specify a different dataset name as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example:

```
vsim -view foo=vsim.wlf
```

ModelSim designates one of the datasets to be the "active" dataset, and refers all names without dataset prefixes to that dataset. The active dataset is displayed in the context path at the bottom of the Main window. When you select a design unit in a dataset's structure tab, that dataset becomes active automatically. Alternatively, you can use the Dataset Browser or the environment command to change the active dataset.

Design regions and signal names can be fully specified over multiple WLF files by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
view:/top/alu/out
golden:.top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open, and you want to refer to something outside the active dataset. When more than one dataset is open, ModelSim will automatically prefix names in the Wave and List windows with the dataset name. You can change this default by selecting **Tools > Window Preferences** (Wave and List windows).

ModelSim also remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the environment command, specifying the dataset without a path. For example:

```
env foo:
```

sets the active dataset to **foo** and the current context to the context last specified for **foo**. The context is then applied to any unlocked windows.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

The Objects pane can be locked to a specific context of a dataset. Being locked to a dataset means that the pane will update only when the content of that dataset changes. If locked to both a dataset and a context (e.g., test: /top/foo), the pane will update only when that specific context changes. You specify the dataset to which the pane is locked by selecting **File > Environment**.

Restricting the Dataset Prefix Display

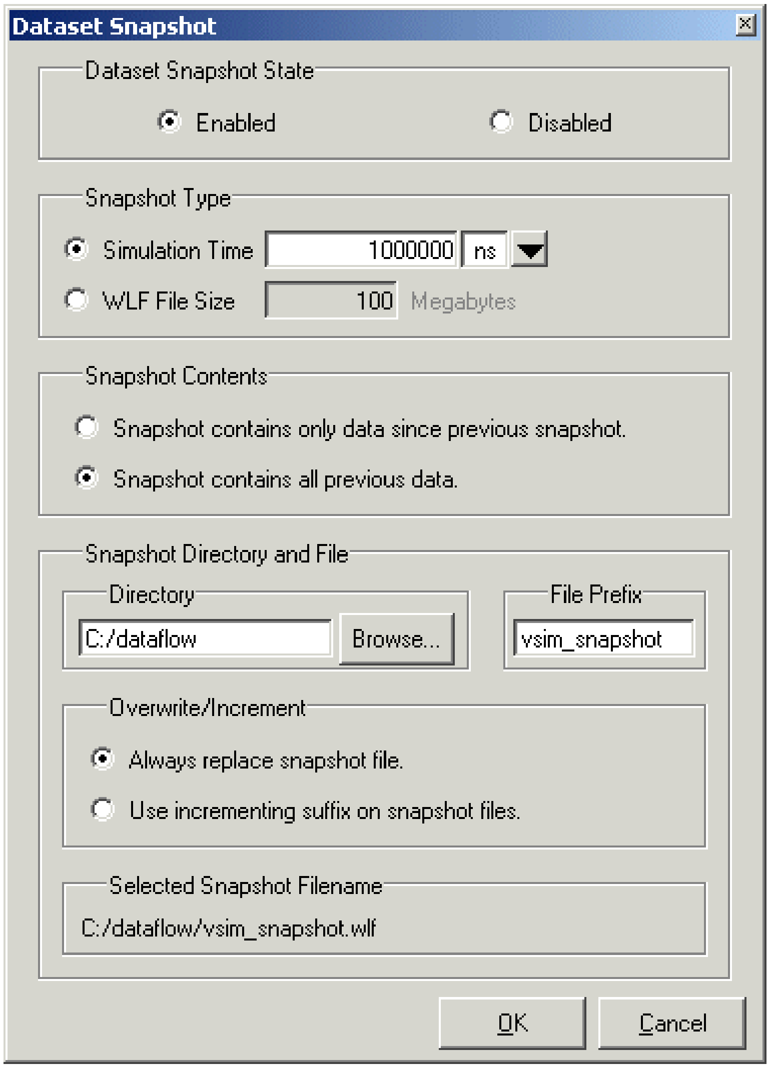
The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. Either edit the *pref.tcl* file directly or use the **Tools > Edit Preferences** command to change the variable value.

Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the environment command with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

Saving at Intervals with Dataset Snapshot

Dataset Snapshot lets you periodically copy data from the current simulation WLF file to another file. This is useful for taking periodic "snapshots" of your simulation or for clearing the current simulation WLF file based on size or elapsed time.

Once you have logged the appropriate objects, select **Tools** > **Dataset Snapshot** (Wave window).



Dataset Snapshot

Dataset Snapshot State

● Enabled ○ Disabled

Snapshot Type

● Simulation Time 1000000 ns

○ WLF File Size 100 Megabytes

Snapshot Contents

○ Snapshot contains only data since previous snapshot.

● Snapshot contains all previous data.

Snapshot Directory and File

Directory: C:/dataflow

File Prefix: vsim_snapshot

Overwrite/Increment

● Always replace snapshot file.

○ Use incrementing suffix on snapshot files.

Selected Snapshot Filename

C:/dataflow/vsim_snapshot.wlf

Collapsing Time and Delta Steps

By default ModelSim collapses delta steps. This means each logged signal that has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. The event order in the WLF file matches the order of the first events of each signal.

You can configure how ModelSim collapses time and delta steps using arguments to the vsim command or by setting the WLFCollapseMode variable in the *modelsim.ini* file. The table below summarizes the arguments and how they affect event recording.

Table 11-3.

vsim argument	effect	modelsim.ini setting
-wlfnocollapse	All events for each logged signal are recorded to the WLF file in the exact order they occur in the simulation.	WLFCollapseMode = 0
-wlfdeltacollapse	Each logged signal which has events during a simulation delta has its final value recorded to the WLF file when the delta has expired. Default.	WLFCollapseMode = 1
-wlftimecollapse	Same as delta collapsing but at the timestep granularity.	WLFCollapseMode = 2

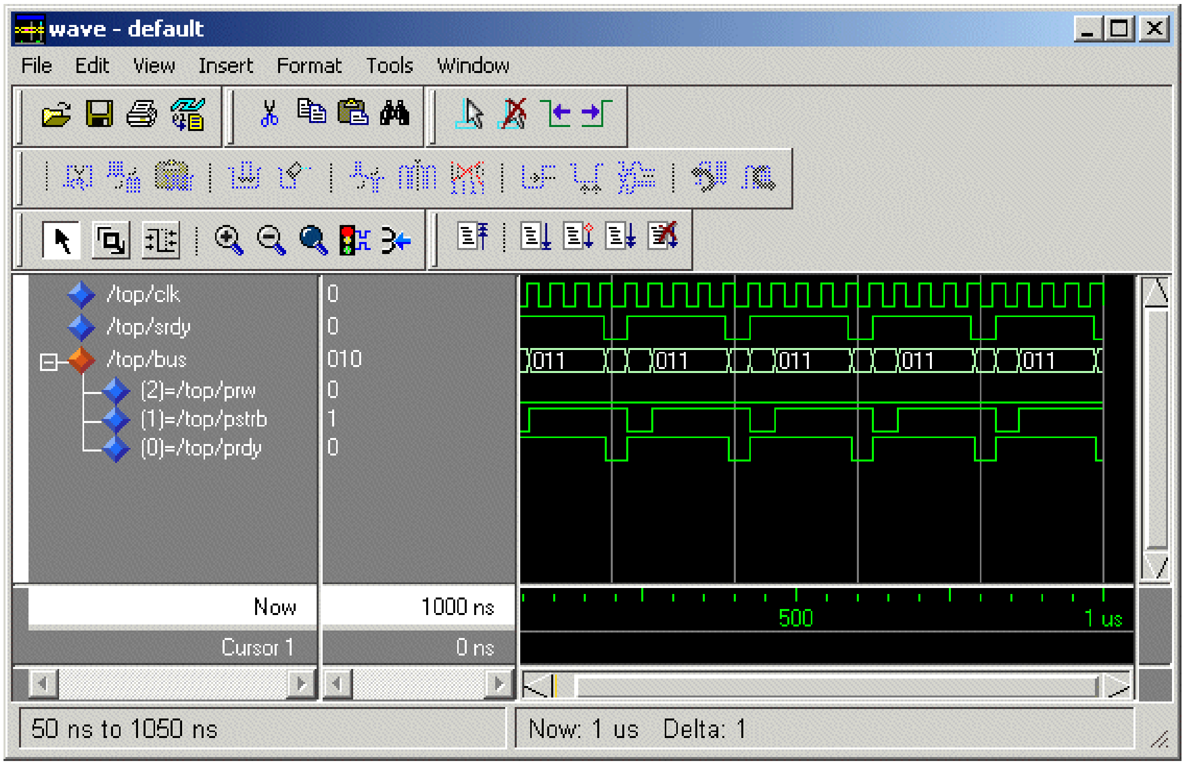
When a run completes that includes single stepping or hitting a breakpoint, all events are flushed to the WLF file regardless of the time collapse mode. It's possible that single stepping through part of a simulation may yield a slightly different WLF file than just running over that piece of code. If particular detail is required in debugging, you should disable time collapsing.

Virtual Objects

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the ModelSim simulation kernel. ModelSim supports the following kinds of virtual objects:

- Virtual Signals
- Virtual Functions
- Virtual Regions
- Virtual Types

Virtual objects are indicated by an orange diamond as illustrated by *bus* below:



Virtual Signals

Virtual signals are aliases for combinations or subelements of signals written to the WLF file by the simulation kernel. They can be displayed in the Objects, List, and Wave windows, accessed by the **examine** command, and set using the **force** command. You can create virtual signals using the **Tools > Combine Signals** (Wave and List windows) menu selections or by using the virtual signal command. Once created, virtual signals can be dragged and dropped from the Objects pane to the Wave and List windows.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual signal should be installed. This can be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The virtual hide command can be used to hide the display of the broken-down bits if you don't want them cluttering up the Objects pane.

If the virtual signal has elements from more than one WLF file, it will be automatically installed in the virtual region *virtuals:/Signals*.

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the scalar elements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the virtual save command. By default, when quitting, ModelSim will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run. There is one exception: "implicit virtuals" are automatically saved with the Wave or List format.

Implicit and Explicit Virtuals

An implicit virtual is a virtual signal that was automatically created by ModelSim without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal whose definition is stored in a special location, and is not visible in the Objects pane or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

Virtual Functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and can be dependent on simulation time. They can be displayed in the Objects, Wave, and List windows and accessed by the examine command, but cannot be set by the force command.

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual function can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the virtual function command.

Virtual functions are also implicitly created by ModelSim when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Objects, Wave, or List window. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

Virtual Regions

User-defined design hierarchy regions can be defined and attached to any existing design region or to the virtuals context tree. They can be used to reconstruct the RTL hierarchy in a gate-level design and to locate virtual signals. Thus, virtual signals and virtual regions can be used in a gate-level design to allow you to use the RTL test bench.

Virtual regions are created and attached using the virtual region command.

Virtual Types

User-defined enumerated types can be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

Virtual types are created using the virtual type command.

Chapter 12
Waveform Analysis

When your simulation finishes, you will often want to analyze waveforms to assess and debug your design. Designers typically use the Wave window for waveform analysis. However, you can also look at waveform data in a textual format in the List window.

To analyze waveforms in ModelSim, follow these steps:

1. Compile your files.
2. Load your design.
3. Add objects to the Wave or List window.

   ```
   add wave <object_name>
   add list <object_name>
   ```

4. Run the design.

Objects You Can View

The list below identifies the types of objects can be viewed in the Wave or List window.

- **VHDL objects** — (indicated by dark blue diamond in the Wave window)

  signals, aliases, process variables, and shared variables
- **Verilog objects** — (indicated by light blue diamond in the Wave window)

  nets, registers, variables, and named events
- **SystemC objects** — (indicated by a green diamond in the Wave window)

  primitive channels and ports
- **Virtual objects** — (indicated by an orange diamond in the Wave window)

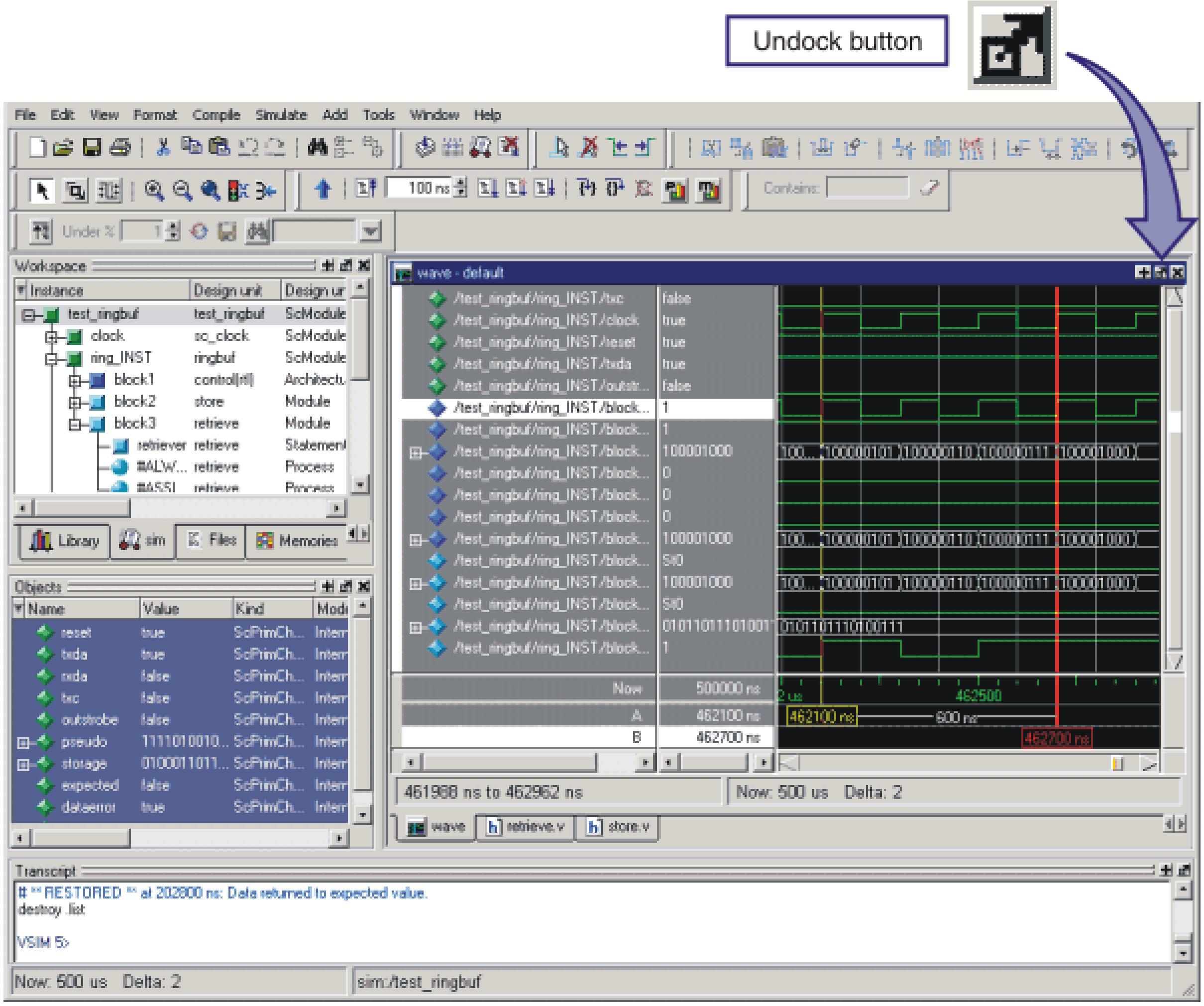
  virtual signals, buses, and functions, see; Virtual Objects for more information
- **Comparisons** — (indicated by a yellow triangle)

  comparison regions and comparison signals; see Waveform Compare for more information

Wave Window Overview

The Wave window opens by default in the MDI frame of the Main window as shown below. The window can be undocked from the main window by pressing the Undock button in the window header or by using the **view -undock wave** command. The preference variable **PrefMain(ViewUnDocked) wave** can be used to control this default behavior. Setting this variable will open the Wave Window undocked each time you start ModelSim.

Figure 12-1. Undocking the Wave window

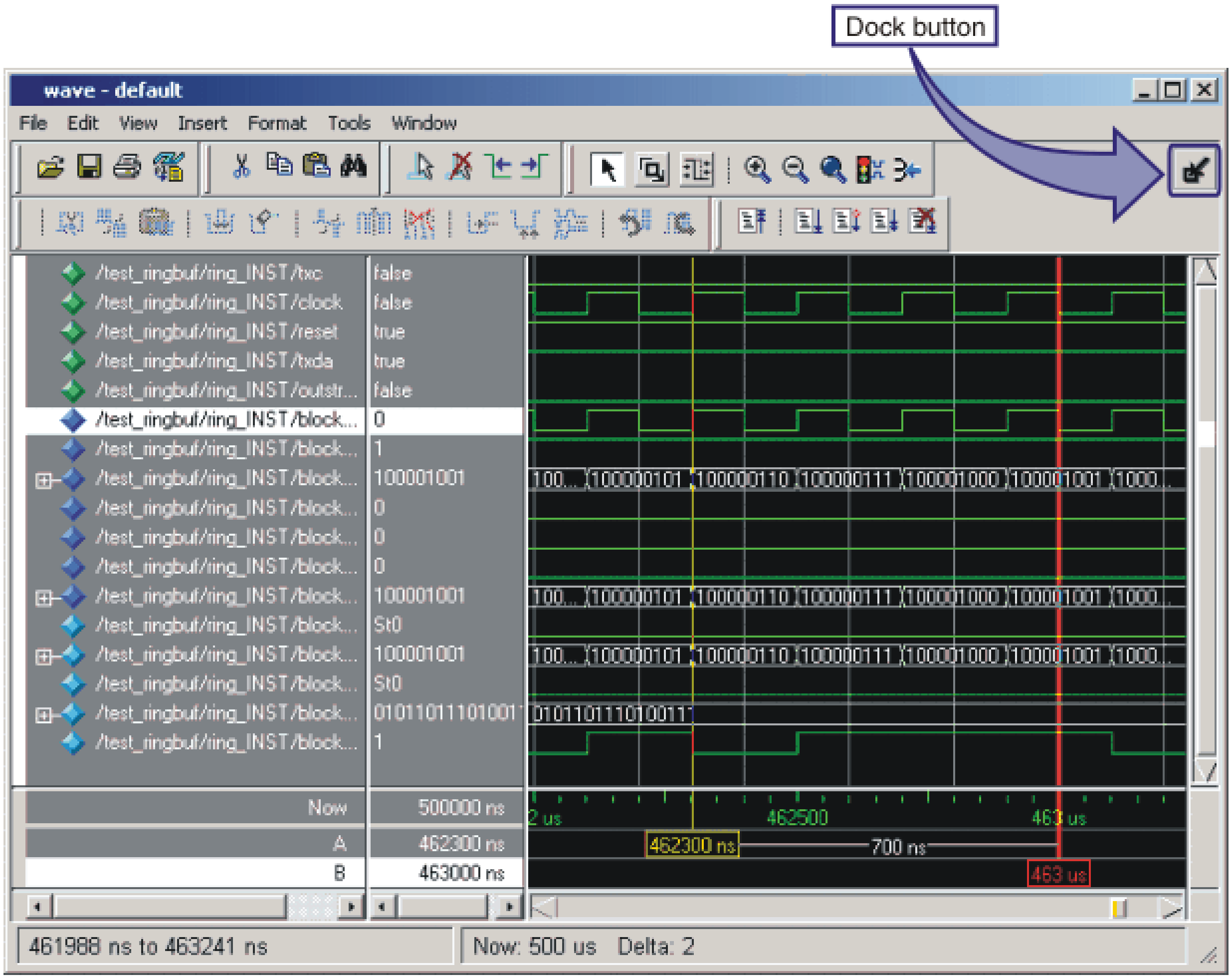


```
# ** RESTORED ** at 202800 ns: Data returned to expected value.
destroy .list

VSIM 5>
```

Now: 500 us   Delta: 2   sim:/test_ringbuf

Here is an example of a Wave window that is undocked from the MDI frame. All menus and icons associated with Wave window functions now appear in the menu and toolbar areas of the Wave window.

Figure 12-2. Docking the Wave window



/test_ringbuf/ring_INST/txc	false
/test_ringbuf/ring_INST/clock	false
/test_ringbuf/ring_INST/reset	true
/test_ringbuf/ring_INST/txda	true
/test_ringbuf/ring_INST/outstr...	false
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	010110111010011
/test_ringbuf/ring_INST/block...	1
Now	500000 ns
A	462300 ns
B	463000 ns

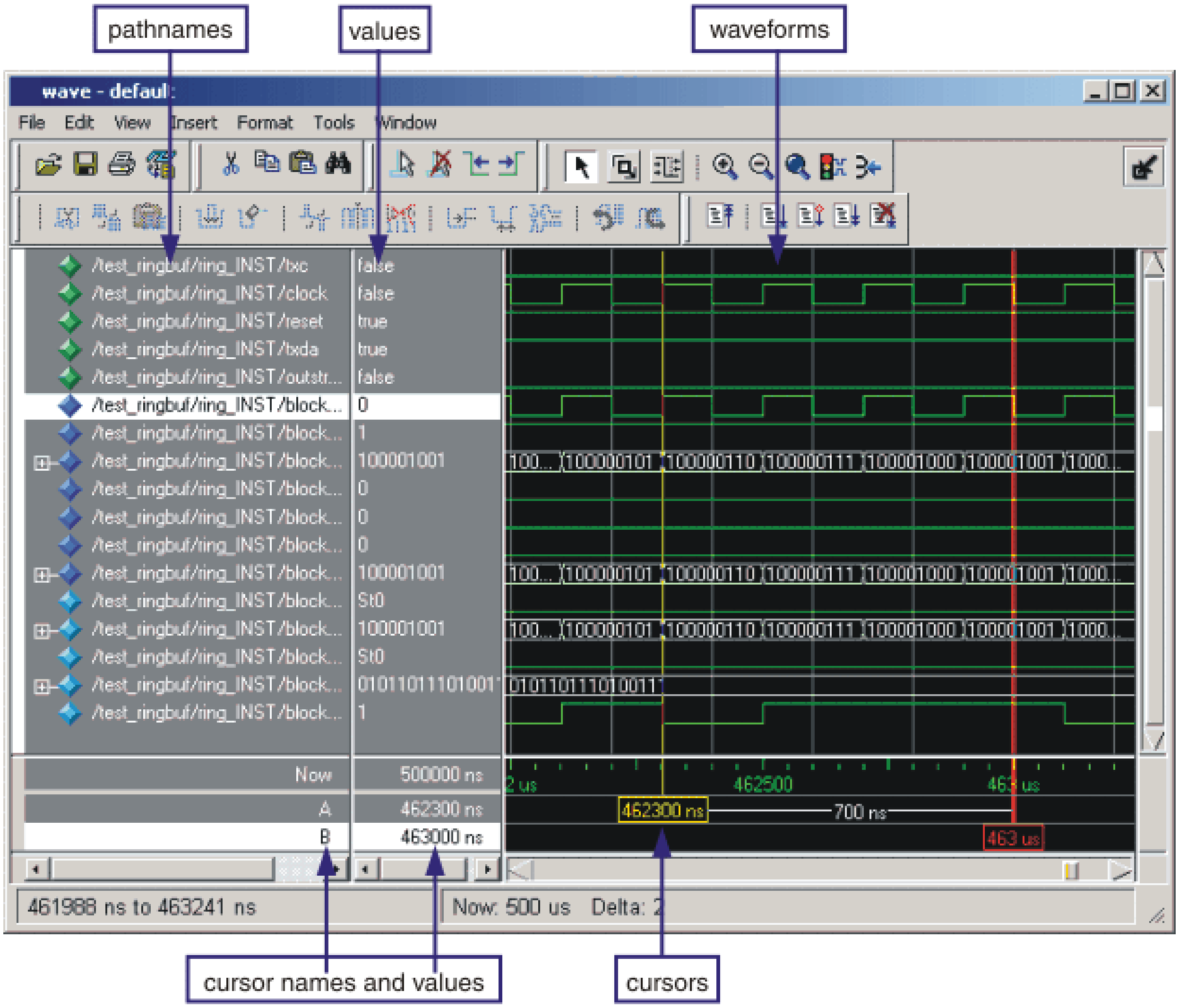
461988 ns to 463241 ns

Now: 500 us Delta: 2

If the Wave window is docked into the Main window MDI frame, all menus and icons that were in the standalone version of the Wave window move into the Main window menu bar and toolbar.

The Wave window is divided into a number of window panes. All window panes in the Wave window can be resized by clicking and dragging the bar between any two panes.

Figure 12-3. Panes in the Wave window



/test_ringbuf/ring_INST/txc	false
/test_ringbuf/ring_INST/clock	false
/test_ringbuf/ring_INST/reset	true
/test_ringbuf/ring_INST/txda	true
/test_ringbuf/ring_INST/outstr...	false
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	0101101110100111
/test_ringbuf/ring_INST/block...	1
Now	500000 ns
A	462300 ns
B	463000 ns

461988 ns to 463241 ns   Now: 500 us   Delta: 2

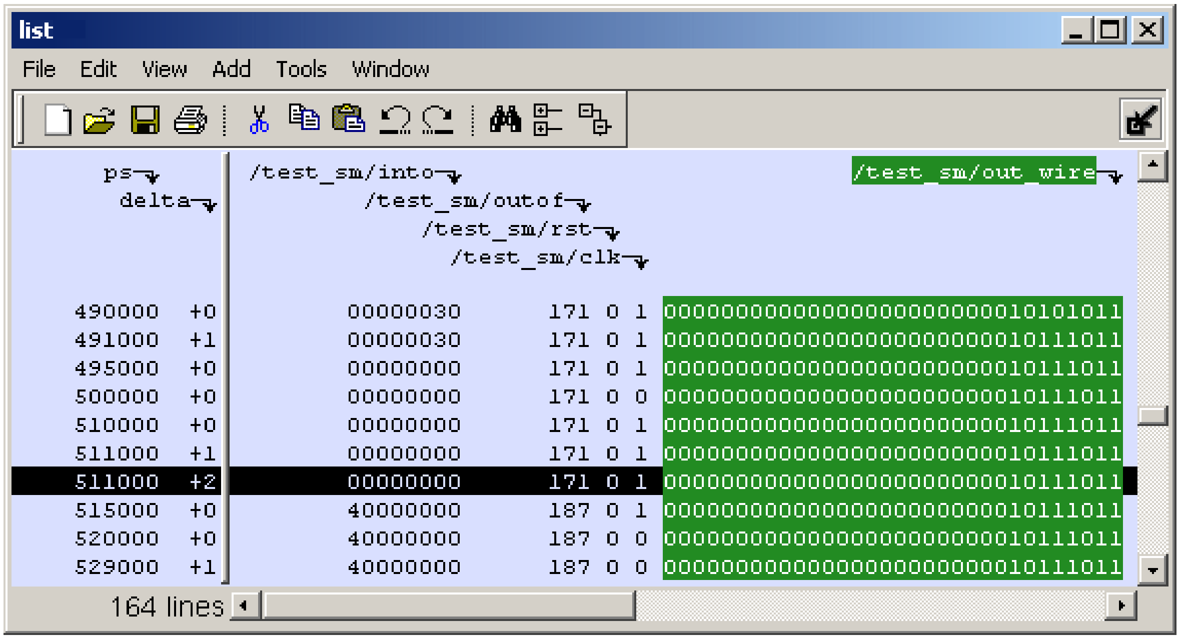
List Window Overview

The List window displays simulation results in tabular format. Common tasks that people use the window for include:

- Using gating expressions and trigger settings to focus in on particular signals or events. See Configuring New Line Triggering in the List Window.
- Debugging delta delay issues. See Delta Delays for more information.

The window is divided into two adjustable panes, which allows you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.

Figure 12-4. The tabular format of the List window



```
          ps     /test_sm/into                               /test_sm/out_wire
          delta           /test_sm/outof
                             /test_sm/rst
                                /test_sm/clk

  490000  +0        00000030     171 0 1  00000000000000000000000010101011
  491000  +1        00000030     171 0 1  00000000000000000000000010111011
  495000  +0        00000000     171 0 1  00000000000000000000000010111011
  500000  +0        00000000     171 0 0  00000000000000000000000010111011
  510000  +0        00000000     171 0 1  00000000000000000000000010111011
  511000  +1        00000000     171 0 1  00000000000000000000000010111011
  511000  +2        00000000     171 0 1  00000000000000000000000010111011
  515000  +0        40000000     187 0 1  00000000000000000000000010111011
  520000  +0        40000000     187 0 0  00000000000000000000000010111011
  529000  +1        40000000     187 0 0  00000000000000000000000010111011
```

164 lines

Adding Objects to the Wave or List Window

You can add objects to the Wave or List window in several ways.

Adding Objects with Drag and Drop

You can drag and drop objects into the Wave or List window from the Workspace, Active Processes, Memory, Objects, Source, or Locals panes. You can also drag objects from the Wave window to the List window and vice versa.

Select the objects in the first window, then drop them into the Wave window. Depending on what you select, all objects or any portion of the design can be added.

Adding Objects with a Menu Command

The **Add** menu in the Main windows let you add objects to the Wave window, List window, or Log file.

Adding Objects with a Command

Use the add list or add wave commands to add objects from the command line. For example:

```
VSIM> add wave /proc/a
```

Adds signal */proc/a* to the Wave window.

VSIM> add list *

Adds all the objects in the current region to the List window.

VSIM> add wave -r /*

Adds all objects in the design to the Wave window.

Adding Objects with a Window Format File

Select **File** > **Open** > **Format** and specify a previously saved format file. See Saving the Window Format for details on how to create a format file.

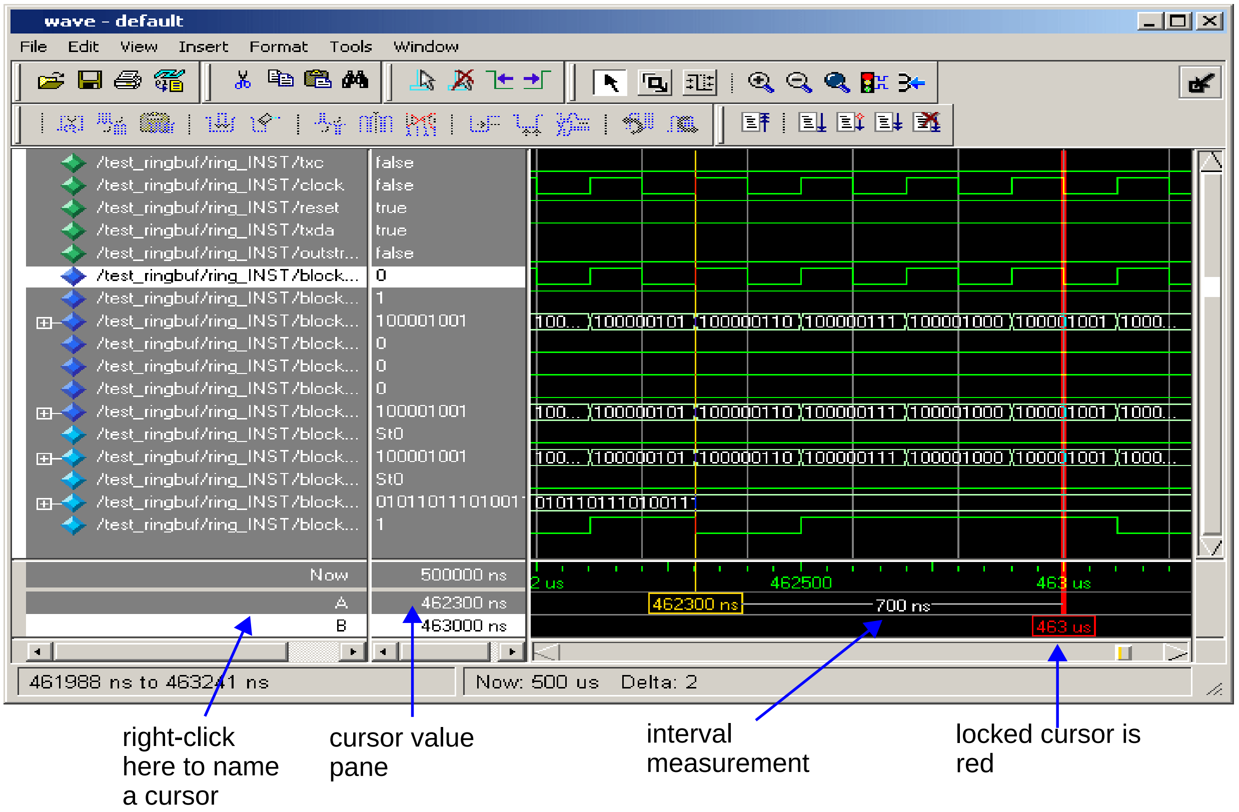
Measuring Time with Cursors in the Wave Window

ModelSim uses cursors to measure time in the Wave window. Cursors extend a vertical line over the waveform display and identify a specific simulation time. Multiple cursors can be used to measure time intervals, as shown in the graphic below.

When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location. The selected cursor is drawn as a bold solid line; all other cursors are drawn with thin lines.

As shown in the graphic below, three window panes relate to cursors: the cursor name pane on the bottom left, the cursor value pane in the bottom middle, and the cursor pane with horizontal "tracks" on the bottom right.

Figure 12-5. Cursor names, values and time measurements



/test_ringbuf/ring_INST/txc	false
/test_ringbuf/ring_INST/clock	false
/test_ringbuf/ring_INST/reset	true
/test_ringbuf/ring_INST/txda	true
/test_ringbuf/ring_INST/outstr...	false
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	1
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	100001001
/test_ringbuf/ring_INST/block...	St0
/test_ringbuf/ring_INST/block...	01011011101001
/test_ringbuf/ring_INST/block...	1
Now	500000 ns
A	462300 ns
B	463000 ns

461988 ns to 463241 ns   Now: 500 us   Delta: 2

Working with Cursors

The table below summarizes common cursor actions.

Table 12-1.

Action	Menu command	Toolbar button
Add cursor	**Insert > Cursor**	
Delete cursor	**Edit > Delete Cursor**	
Zoom In on Active Cursor	**View > Zoom > Zoom Cursor**	
Lock cursor	**Edit > Edit Cursor**	NA
Name cursor	**Edit > Edit Cursor**	NA

Table 12-1.

Action	Menu command	Toolbar button
Select cursor	**View > Cursors**	NA

Shortcuts for Working with Cursors

There are a number of useful keyboard and mouse shortcuts related to the actions listed above:

- Select a cursor by clicking the cursor name.
- Jump to a "hidden" cursor (one that is out of view) by double-clicking the cursor name.
- Name a cursor by right-clicking the cursor name and entering a new value. Press <Enter> on your keyboard after you have typed the new name.
- Move a locked cursor by holding down the <shift> key and then clicking-and-dragging the cursor.
- Move a cursor to a particular time by right-clicking the cursor value and typing the value to which you want to scroll. Press <Enter> on your keyboard after you have typed the new value.

Understanding Cursor Behavior

The following list describes how cursors "behave" when you click in various panes of the Wave window:

- If you click in the waveform pane, the cursor closest to the mouse position is selected and then moved to the mouse position.
- Clicking in a horizontal "track" in the cursor pane selects that cursor and moves it to the mouse position.
- Cursors "snap" to a waveform edge if you click or drag a cursor along the selected waveform to within ten pixels of a waveform edge. You can set the snap distance in the Window Preferences dialog. Select **Tools > Options > Wave Preferences** when the Wave window is docked in the Main window MDI frame. Select **Tools > Window Preferences** when the Wave window is a stand-alone, undocked window.
- You can position a cursor without snapping by dragging in the cursor pane below the waveforms.

Jumping to a Signal Transition

You can move the active cursor to the next or previous transition on the selected signal using these two buttons on the toolbar:

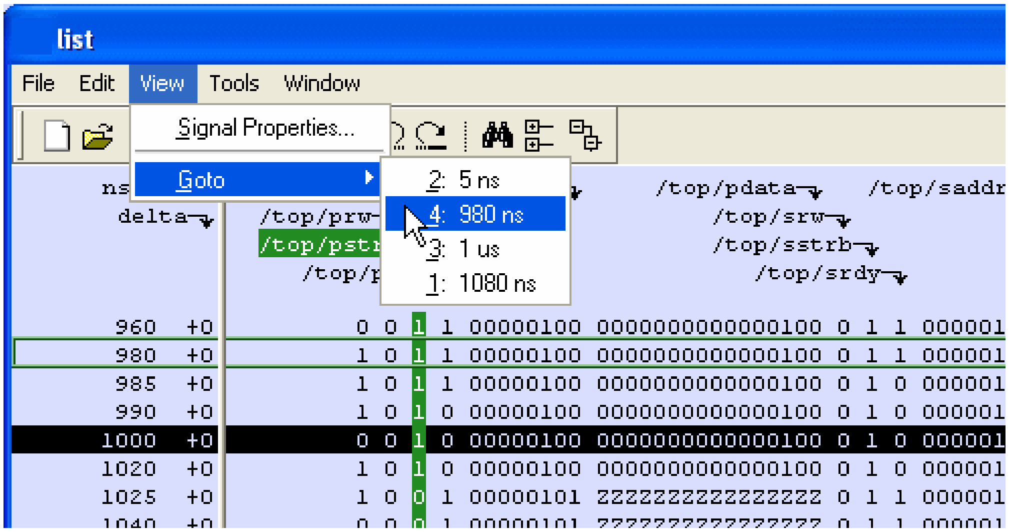
Find Previous Transition locate the previous signal value change for the selected signal

Find Next Transition locate the next signal value change for the selected signal

Setting Time Markers in the List Window

Time markers in the List window are similar to cursors in the Wave window. Time markers tag lines in the data table so you can quickly jump back to that time. Markers are indicated by a thin box surrounding the marked line.

Figure 12-6. Time markers in the List window



```
 960  +0    0 0 1 1 00000100 0000000000000100 0 1 1 000001
 980  +0    1 0 1 1 00000100 0000000000000100 0 1 1 000001
 985  +0    1 0 1 1 00000100 0000000000000100 0 1 0 000001
 990  +0    1 0 1 0 00000100 0000000000000100 0 1 0 000001
1000  +0    0 0 1 0 00000100 0000000000000100 0 1 0 000001
1020  +0    1 0 1 0 00000100 0000000000000100 0 1 0 000001
1025  +0    1 0 0 1 00000101 ZZZZZZZZZZZZZZZZ 0 1 1 000001
1040  +0    0 0 0 1 00000101 ZZZZZZZZZZZZZZZZ 0 1 1 000001
```

Working with Markers

The table below summarizes actions you can take with markers.

Table 12-2.

Action	Method
Add marker	Select a line and then select **Edit** > **Add Marker**
Delete marker	Select a tagged line and then select **Edit** > **Delete Marker**
Goto marker	Select **View** > **Goto** > **<time>**

Zooming the Wave Window Display

Zooming lets you change the simulation range in the waveform pane. You can zoom using the context menu, toolbar buttons, mouse, keyboard, or commands.

Zooming with the Menu, Toolbar and Mouse

You can access Zoom commands from the **View** menu on the toolbar or by clicking the right mouse button in the waveform pane.

These zoom buttons are available on the toolbar:

Zoom In 2x
zoom in by a factor of two from the current view

Zoom Out 2x
zoom out by a factor of two from current view

Zoom In on Active Cursor
centers the active cursor in the waveform display and zooms in

Zoom Full
zoom out to view the full range of the simulation from time 0 to the current time

Zoom Mode
change mouse pointer to zoom mode; see below

To zoom with the mouse, first enter zoom mode by selecting **View** > **Zoom** > **Mouse Mode** > **Zoom Mode**. The left mouse button then offers 3 zoom options by clicking and dragging in different directions:

- Down-Right *or* Down-Left: Zoom Area (In)
- Up-Right: Zoom Out
- Up-Left: Zoom Fit

Also note the following about zooming with the mouse:

- The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.
- You can enter zoom mode temporarily by holding the <Ctrl> key down while in select mode.
- With the mouse in the Select Mode, the middle mouse button will perform the above zoom operations.

Saving Zoom Range and Scroll Position with Bookmarks

Bookmarks save a particular zoom range and scroll position. This lets you return easily to a specific view later. You save the bookmark with a name and then access the named bookmark from the Bookmark menu. Bookmarks are saved in the Wave format file (see Adding Objects with a Window Format File) and are restored when the format file is read.

Managing Bookmarks

The table below summarizes actions you can take with bookmarks.

Table 12-3.

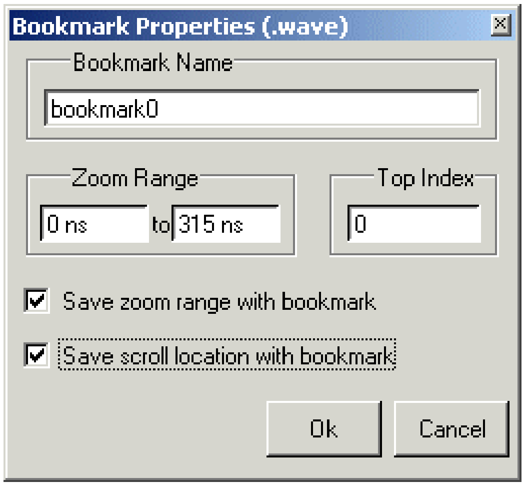
Action	Menu	Command
Add bookmark	**Edit > Insert Bookmark**	bookmark add wave
View bookmark	**View > Bookmark > <name>**	bookmark goto wave
Delete bookmark	**Tools > Bookmarks**	bookmark delete wave

Adding Bookmarks

To add a bookmark, follow these steps:

1. Zoom the wave window as you see fit using one of the techniques discussed in Zooming the Wave Window Display.
2. Select **Edit > Insert Bookmark**.

Figure 12-7. Bookmark Properties dialog



Bookmark Properties (.wave)

Bookmark Name: bookmark0

Zoom Range: 0 ns to 315 ns

Top Index: 0

☑ Save zoom range with bookmark

☑ Save scroll location with bookmark

3. Give the bookmark a name and click OK.

Editing Bookmarks

Once a bookmark exists, you can change its properties by selecting **Tools** > **Bookmarks**.

Searching in the Wave and List Windows

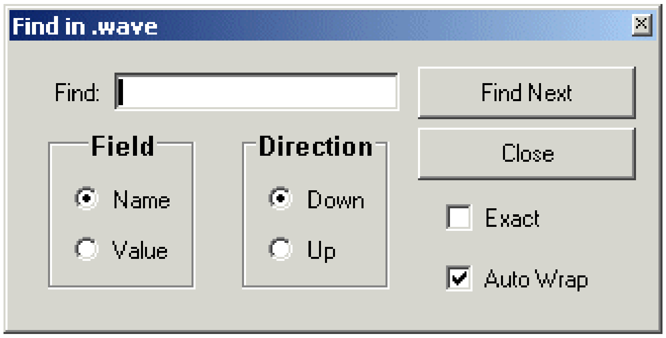
The Wave and List windows provide two methods for locating objects:

- Finding signal names – Select **Edit** > **Find** or use the find command to search for the name of a signal.
- Search for values or transitions – Select **Edit** > **Search** or use the search command to locate transitions or signal values. The search feature is not available in all versions of ModelSim.

Finding Signal Names

The Find command is used to locate a signal name or value in the Wave or List window. When you select **Edit** > **Find**, the Find dialog appears.

Figure 12-8. Find signals by name or value



Find:

Field
● Name
○ Value

Direction
● Down
○ Up

□ Exact
☑ Auto Wrap

One option of note is the "Exact" checkbox. Check **Exact** if you only want to find objects that match your search exactly. For example, searching for "clk" without **Exact** will find */top/clk* and *clk1*.

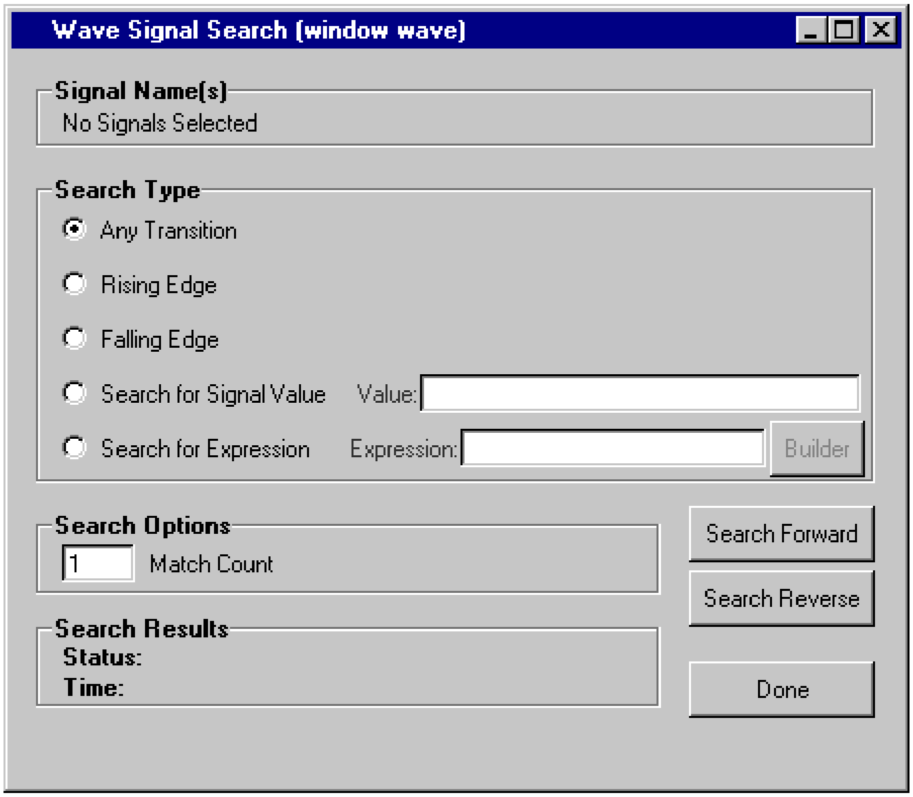
There are two differences between the Wave and List windows as it relates to the Find feature:

- In the Wave window you can specify a value to search for in the values pane.
- The find operation works only within the active pane in the Wave window.

Searching for Values or Transitions

Available in some versions of ModelSim, the Search command lets you search for transitions or values on selected signals. When you select **Edit** > **Search**, the Signal Search dialog appears.

Figure 12-9. Wave Signal Search dialog



Wave Signal Search (window wave)

Signal Name(s)
No Signals Selected

Search Type
● Any Transition
○ Rising Edge
○ Falling Edge
○ Search for Signal Value Value:
○ Search for Expression Expression:

Search Options
1 Match Count

Search Results
Status:
Time:

One option of note is **Search for Expression**. The expression can involve more than one signal but is limited to signals currently in the window. Expressions can include constants, variables, and DO files. See Expression Syntax for more information.

Note

If your signal values are displayed in binary radix, see Searching for Binary Signal Values in the GUI for details on how signal values are mapped between a binary radix and std_logic.

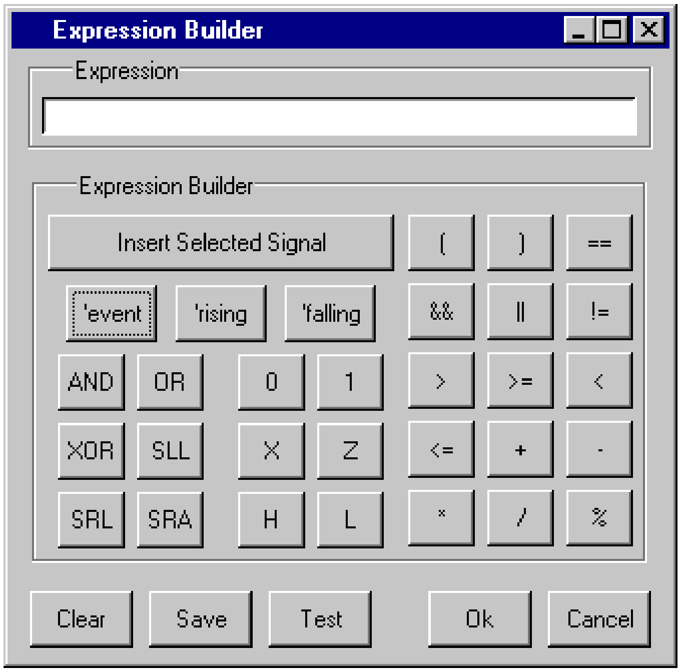
Using the Expression Builder for Expression Searches

The Expression Builder is a feature of the Wave and List Signal Search dialog boxes, and the List trigger properties dialog box. It aids in building a search expression that follows the GUI_expression_format.

To locate the Builder:

- select **Edit** > **Search** (List or Wave window)
- select the **Search for Expression** option in the resulting dialog box
- select the **Builder** button

Figure 12-10. Expression Builder dialog



The Expression Builder dialog box provides an array of buttons that help you build a GUI expression. For instance, rather than typing in a signal name, you can select the signal in the associated Wave or List window and press Insert Selected Signal. All Expression Builder buttons correspond to the Expression Syntax.

Saving an Expression to a Tcl Variable

Clicking the **Save** button will save the expression to a Tcl variable. Once saved this variable can be used in place of the expression. For example, say you save an expression to the variable "foo". Here are some operations you could do with the saved variable:

- Read the value of *foo* with the set command:

  ```
  set foo
  ```

- Put $foo in the Expression: entry box for the Search for Expression selection.
- Issue a searchlog command using foo:

  ```
  searchlog -expr $foo 0
  ```

Searching for when a Signal Reaches a Particular Value

Select the signal in the Wave window and click **Insert Selected Signal** and ==. Then, click the value buttons or type a value.

Evaluating Only on Clock Edges

Click the **&&** button to AND this condition with the rest of the expression. Then select the clock in the Wave window and click **Insert Selected Signal** and **‘rising**. You can also select the falling edge or both edges.

Operators

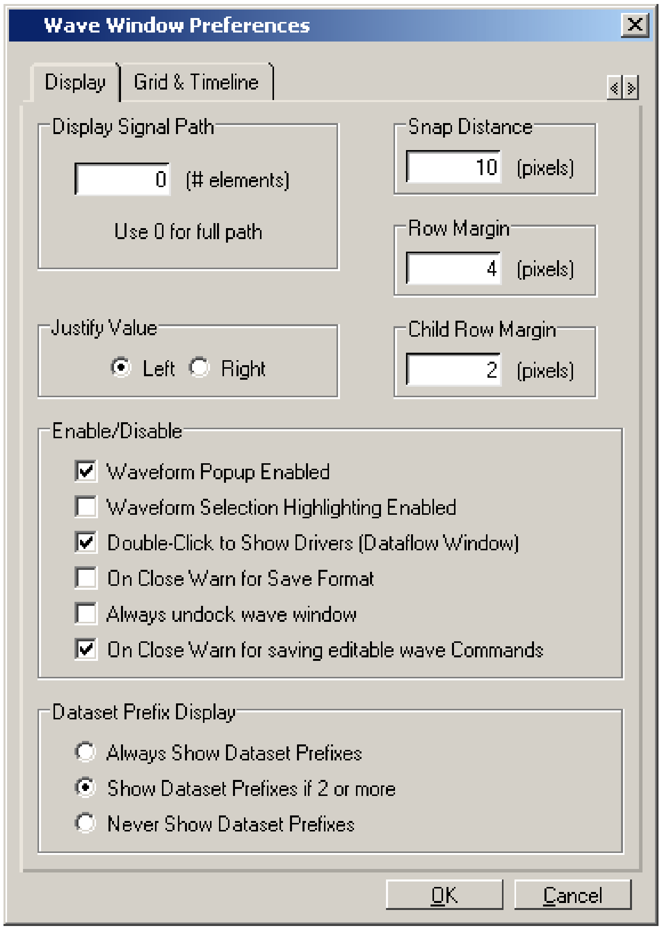
Other buttons will add operators of various kinds (see Expression Syntax), or you can type them in.

Formatting the Wave Window

Setting Wave Window Display Preferences

You can set Wave Window display preferences by selecting **Tools** > **Options** > **Wave Preferences** (when the window is docked in the MDI frame) or **Tools** > **Window Preferences** (when the window is undocked). These commands open the Wave Window Preferences dialog.

Figure 12-11. Display tab of the Wave Window Preferences dialog



Wave Window Preferences

Display | Grid & Timeline

Display Signal Path
0 (# elements)
Use 0 for full path

Snap Distance
10 (pixels)

Row Margin
4 (pixels)

Justify Value
● Left ○ Right

Child Row Margin
2 (pixels)

Enable/Disable
☑ Waveform Popup Enabled
☐ Waveform Selection Highlighting Enabled
☑ Double-Click to Show Drivers (Dataflow Window)
☐ On Close Warn for Save Format
☐ Always undock wave window
☑ On Close Warn for saving editable wave Commands

Dataset Prefix Display
○ Always Show Dataset Prefixes
● Show Dataset Prefixes if 2 or more
○ Never Show Dataset Prefixes

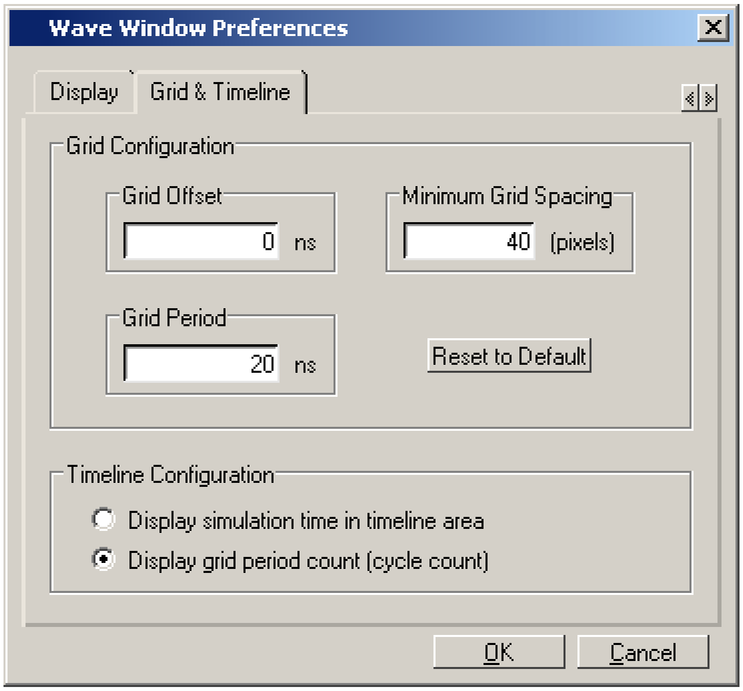
Hiding/Showing Path Hierarchy

You can set how many elements of the object path display by changing the Display Signal Path value in the Wave Window Preferences dialog (Figure 12-11). Zero indicates the full path while a non-zero number indicates the number of path elements to be displayed.

Setting the Timeline to Count Clock Cycles

You can set the timeline of the Wave window to count clock cycles rather than elapsed time. If the Wave window is docked in the MDI frame, open the Wave Window Preferences dialog by selecting Tools > Options > Wave Preferences from the Main window menus. If the Wave window is undocked, select Tools > Window Preferences from the Wave window menus. This opens the Wave Window Preferences dialog. In the dialog, select the Grid & Timeline tab (Figure 12-12).

Figure 12-12. The Grid & Timeline tab of the Wave Window Preferences dialog



Grid Configuration

Grid Offset: 0 ns

Minimum Grid Spacing: 40 (pixels)

Grid Period: 20 ns

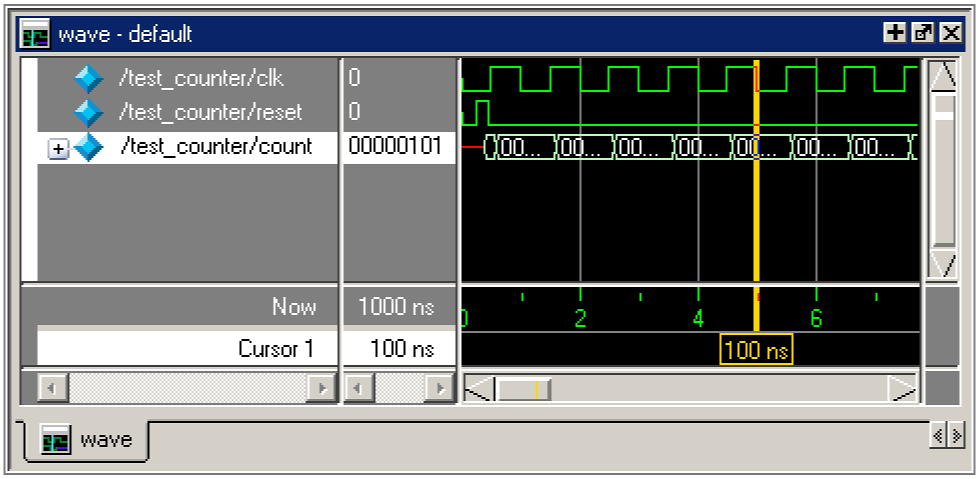
Timeline Configuration

○ Display simulation time in timeline area

● Display grid period count (cycle count)

Enter the period of your clock in the Grid Period field and select “Display grid period count (cycle count).” The timeline will now show the number of clock cycles, as shown in .

Figure 12-13. Clock cycles in the timeline of the Wave window



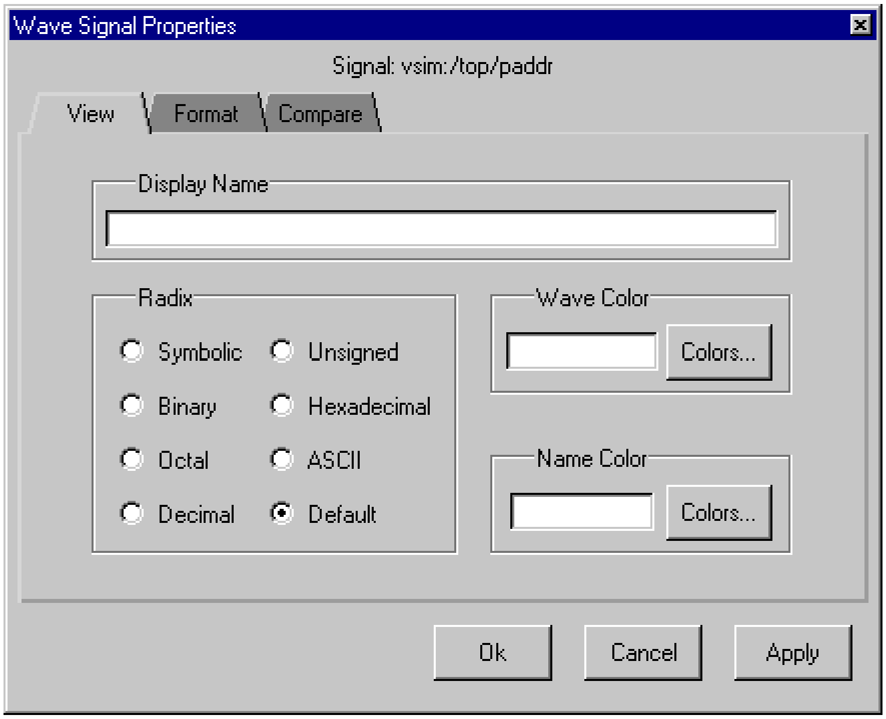
Formatting Objects in the Wave Window

You can adjust various object properties to create the view you find most useful. Select one or more objects and then select **View** > **Properties** or use the selections in the **Format** menu.

Changing Radix (base) for the Wave Window

One common adjustment is changing the radix (base) of an object. When you select **View** > **Properties**, the Wave Signal Properties dialog appears.

Figure 12-14. Changing signal radix



Signal: vsim:/top/paddr

Radix

○ Symbolic ○ Unsigned
○ Binary ○ Hexadecimal
○ Octal ○ ASCII
○ Decimal ● Default

The default radix is symbolic, which means that for an enumerated type, the value pane lists the actual values of the enumerated type of that object. For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Note

When the symbolic radix is chosen for SystemVerilog reg and integer types, the values are treated as binary. When the symbolic radix is chosen for SystemVerilog bit and int types, the values are considered to be decimal.

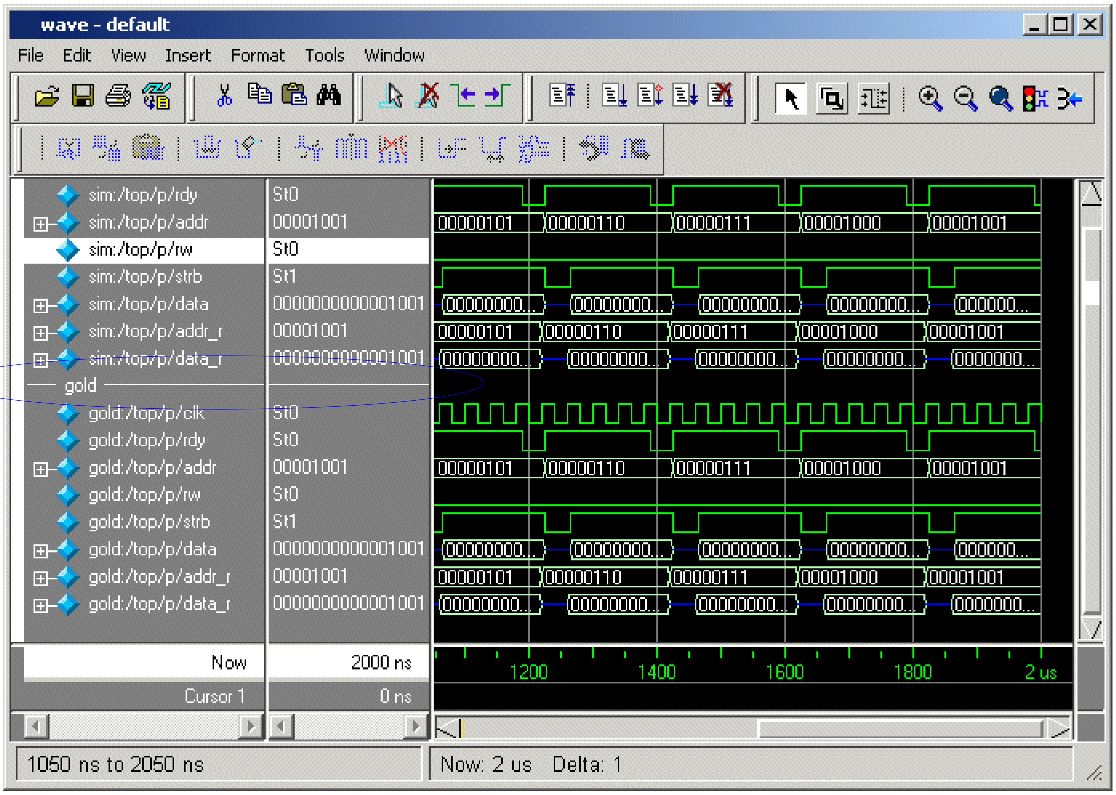
Aside from the Wave Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate** > **Runtime Options** (Main window)
- Change the default radix for the current simulation using the radix command.
- Change the default radix permanently by editing the DefaultRadix variable in the *modelsim.ini* file.

Dividing the Wave Window

Dividers serve as a visual aid for debugging, allowing you to separate signals and waveforms for easier viewing. In the graphic below, two datasets have been separated with a divider called "gold."

Figure 12-15. Wave window dividers allow you to separate signals



sim:/top/p/rdy	St0
sim:/top/p/addr	00001001
sim:/top/p/rw	St0
sim:/top/p/strb	St1
sim:/top/p/data	0000000000001001
sim:/top/p/addr_r	00001001
sim:/top/p/data_r	0000000000001001
gold	
gold:/top/p/clk	St0
gold:/top/p/rdy	St0
gold:/top/p/addr	00001001
gold:/top/p/rw	St0
gold:/top/p/strb	St1
gold:/top/p/data	0000000000001001
gold:/top/p/addr_r	00001001
gold:/top/p/data_r	0000000000001001
Now	2000 ns
Cursor 1	0 ns

1050 ns to 2050 ns

Now: 2 us Delta: 1

To insert a divider, follow these steps:

1. Select the signal above which you want to place the divider.
2. If the Wave pane is docked in MDI frame of the Main window, select **Add** > **Divider** from the Main window menu bar. If the Wave window stands alone, undocked from the Main window, select **Insert** > **Divider** from the Wave window menu bar.
3. Specify the divider name in the Wave Divider Properties dialog. The default name is New Divider. Unnamed dividers are permitted. Simply delete "New Divider" in the Divider Name field to create an unnamed divider.
4. Specify the divider height (default height is 17 pixels) and then click OK.

You can also insert dividers with the **-divider** argument to the add wave command.

Working with Dividers

The table below summarizes several actions you can take with dividers:

Table 12-4.

Action	Method
Move a divider	Click-and-drag the divider to the desired location
Change a divider’s name or size	Right-click the divider and select Divider Properties
Delete a divider	Right-click the divider and select Delete

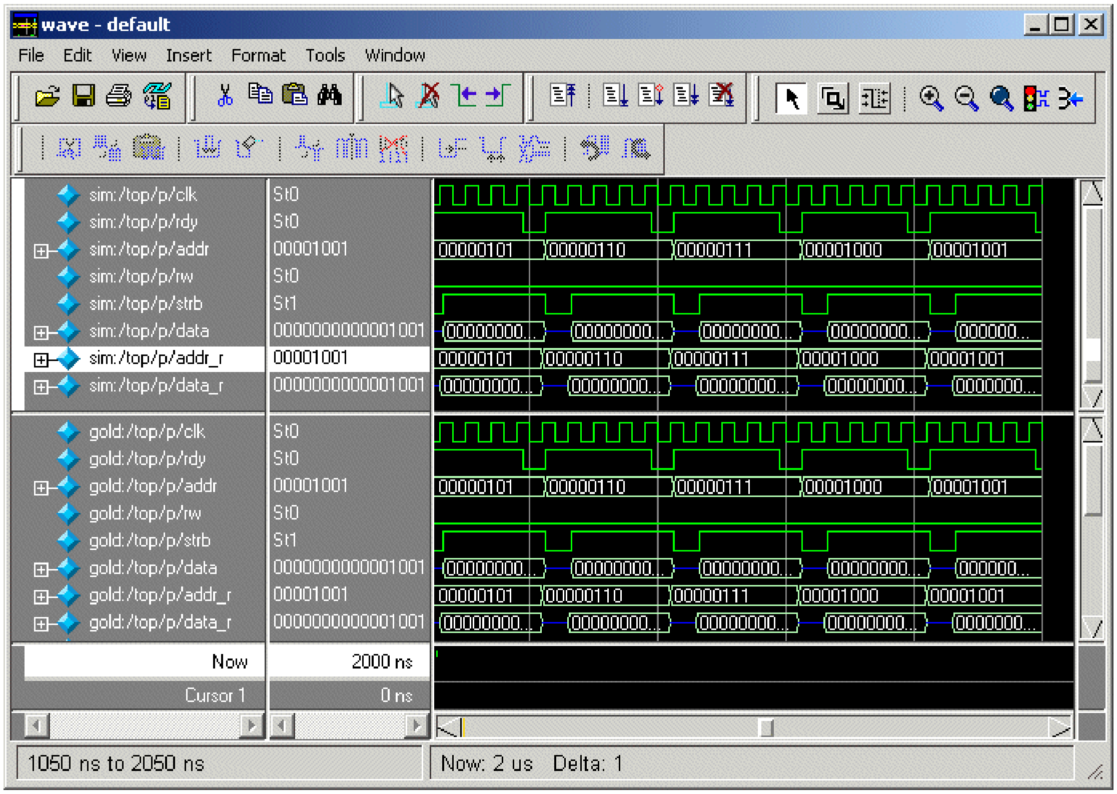
Splitting Wave Window Panes

The pathnames, values, and waveforms panes of the Wave window display can be split to accommodate signals from one or more datasets. For more information on viewing multiple simulations, see WLF Files (Datasets) and Virtuals.

To split the window, select **Insert > Window Pane**.

In the illustration below, the top split shows the current active simulation with the prefix "sim," and the bottom split shows a second dataset with the prefix "gold".

Figure 12-16. Splitting Wave window panes



sim:/top/p/clk	St0
sim:/top/p/rdy	St0
sim:/top/p/addr	00001001
sim:/top/p/rw	St0
sim:/top/p/strb	St1
sim:/top/p/data	0000000000001001
sim:/top/p/addr_r	00001001
sim:/top/p/data_r	0000000000001001
gold:/top/p/clk	St0
gold:/top/p/rdy	St0
gold:/top/p/addr	00001001
gold:/top/p/rw	St0
gold:/top/p/strb	St1
gold:/top/p/data	0000000000001001
gold:/top/p/addr_r	00001001
gold:/top/p/data_r	0000000000001001
Now	2000 ns
Cursor 1	0 ns

1050 ns to 2050 ns

Now: 2 us Delta: 1

The Active Split

The active split is denoted with a solid white bar to the left of the signal names. The active split becomes the target for objects added to the Wave window.

Wave Groups

Wave groups are a wave window specific container object for creating arbitrary groups of items. A wave group may contain 0, 1 or many items. The command line as well as drag and drop may be used to add or remove items from a group. Groups themselves may be dragged around the wave window or to another wave window.

Currently, groups may not be nested.

Creating a Wave Group

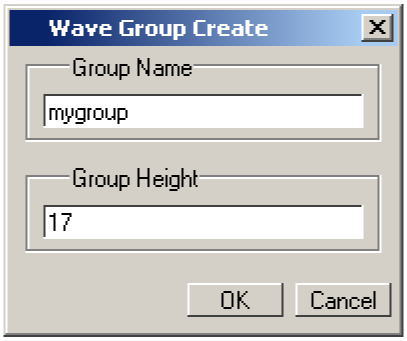
There are two ways to create a wave group.

1. Use the **Tools** > **Group** menu selection.

   a. Select a set of signals in the wave window.

   b. Select the **Tools** > **Group** menu item. The Wave Group Create dialog will appear.

Figure 12-17. Fill in the name of the group in the Group Name field.

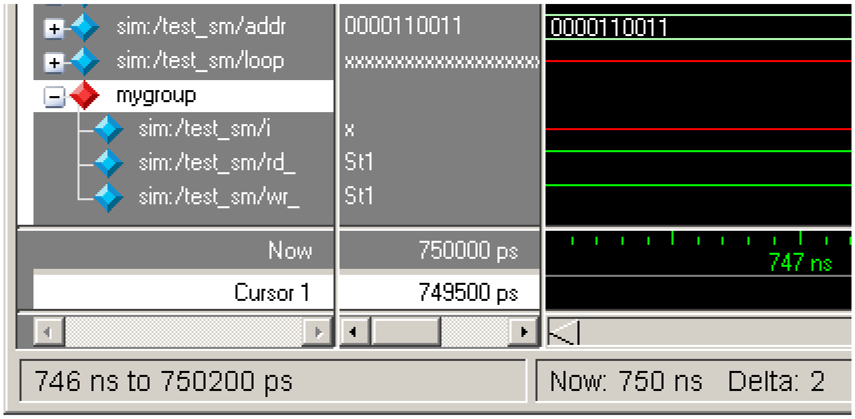


Group Name: mygroup

Group Height: 17

   c. Click Ok. The new wave group will be denoted by a red diamond in the Wave window pathnames.

Figure 12-18. Wave groups denoted by red diamond



2. Use the -group argument to the add wave command.

   Example 1 — The following command will create a group named *mygroup* containing three items:

   ```
   add wave -group mygroup sig1 sig2 sig3
   ```

   Example 2 — The following command will create an empty group named *mygroup*:

   ```
   add wave -group mygroup
   ```

Deleting or Ungrouping a Wave Group

If a wave group is selected and cut or deleted the entire group and all its contents will be removed from the wave window. Likewise, the delete wave command will remove the entire group if the group name is specified.

If a wave group is selected and the **Tools** > **Ungroup** menu item is selected the group will be removed and all of its contents will remain in the Wave window in existing order.

Adding Items to an Existing Wave Group

There are three ways to add items to an existing wave group.

1. Using the drag and drop capability to move items outside of the group or from other windows within ModelSim into the group. The insertion indicator will show the position the item will be dropped into the group. If the cursor is moved over the lower portion of the group item name a box will be drawn around the group name indicating the item will be dropped into the last position in the group.
2. The cut/copy/paste functions may be used to paste items into a group.

3. Use the **add wave -group** command.

   The following example adds two more signals to an existing group called *mygroup*.

   ```
   add wave -group mygroup sig4 sig5
   ```

Removing Items from an Existing Wave Group

You can use any of the following methods to remove an item from a wave group.

1. Use the drag and drop capability to move an item outside of the group.
2. Use menu or icon selections to cut or delete an item or items from the group.
3. Use the delete wave command to specify a signal to be removed from the group.

Note

The delete wave command removes all occurrances of a specified name from the wave window, not just an occurrance within a group.

Miscellaneous Wave Group Features

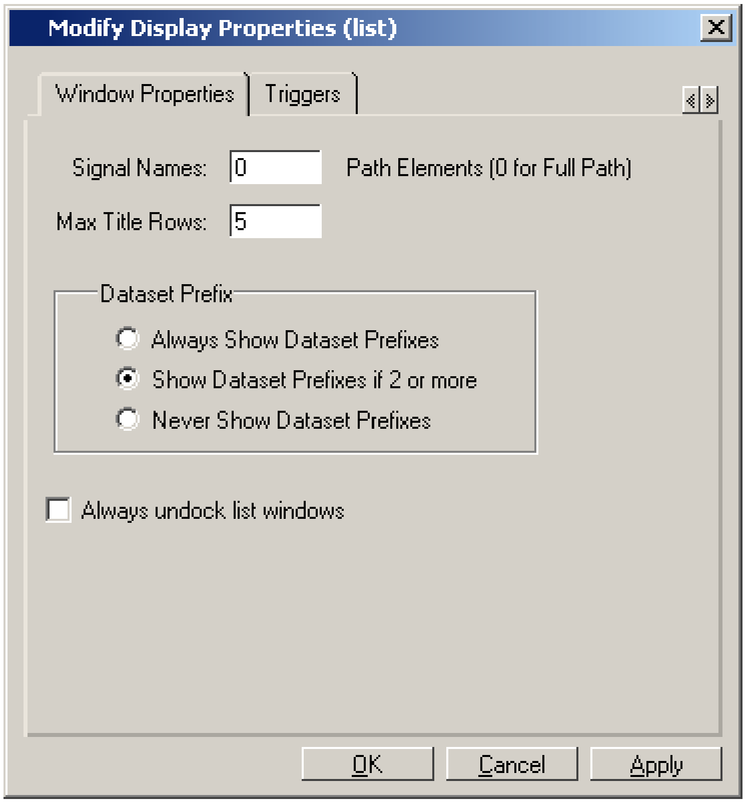
Dragging a wave group from the Wave window to the List window will result in all of the items within the group being added to the List window.

Dragging a group from the Wave window to the Transcript window will result in a list of all of the items within the group being added to the existing command line, if any.

Formatting the List Window

Setting List Window Display Properties

Before you add objects to the List window, you can set the window's display properties. To change when and how a signal is displayed in the List window, select **Tools** > **Window Preferences** from the List window menu bar (when the window is undocked).



Signal Names: 0 Path Elements (0 for Full Path)

Max Title Rows: 5

Dataset Prefix

○ Always Show Dataset Prefixes
● Show Dataset Prefixes if 2 or more
○ Never Show Dataset Prefixes

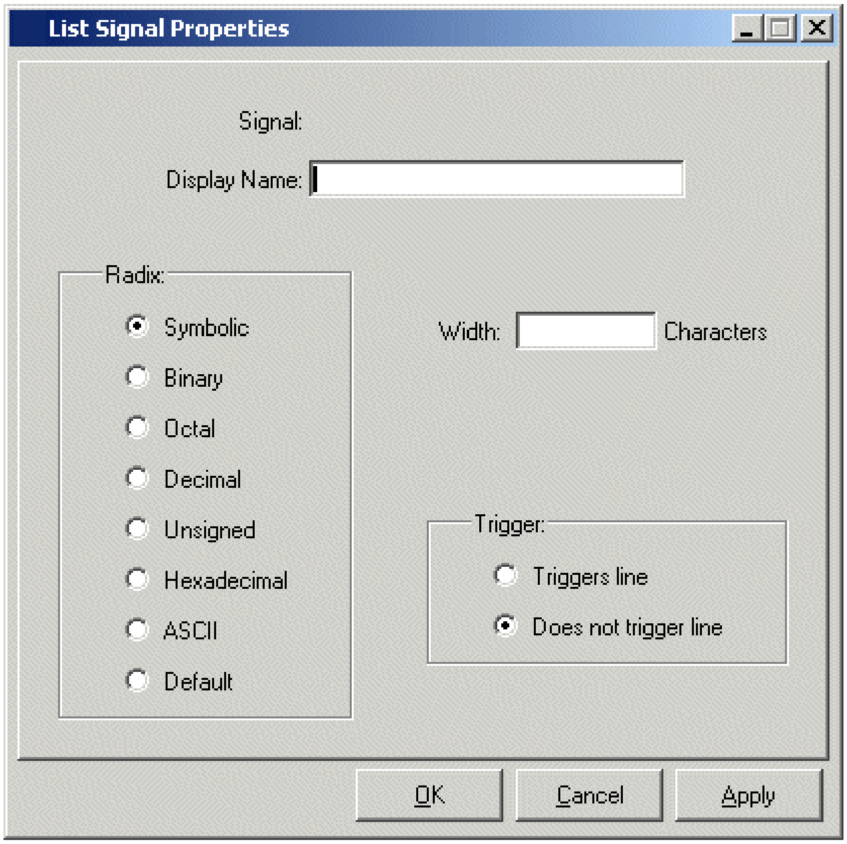
□ Always undock list windows

Formatting Objects in the List Window

You can adjust various properties of objects to create the view you find most useful. Select one or more objects and then select **View** > **Signal Properties** from the List window menu bar (when the window is undocked).

Changing Radix (base) for the List Window

One common adjustment is changing the radix (base) of an object. When you select **View** > **Signal Properties**, the List Signal Properties dialog appears:



List Signal Properties

Signal:

Display Name:

Radix:

● Symbolic
○ Binary
○ Octal
○ Decimal
○ Unsigned
○ Hexadecimal
○ ASCII
○ Default

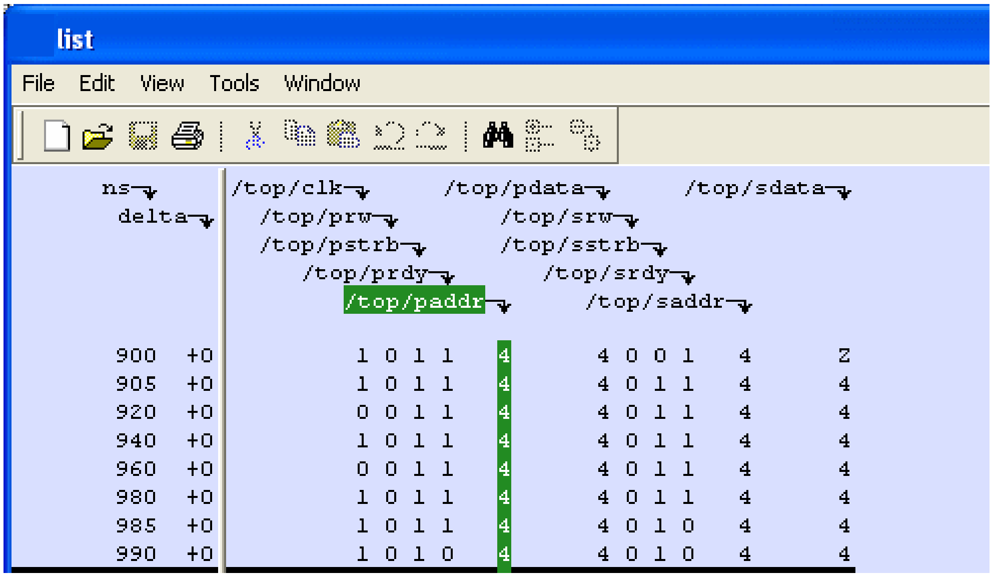
Width: Characters

Trigger:

○ Triggers line
● Does not trigger line

The default radix is symbolic, which means that for an enumerated type, the window lists the actual values of the enumerated type of that object. For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the object value is converted to an appropriate representation in that radix.

Changing the radix can make it easier to view information in the List window. Compare the image below (with decimal values) with the image in the section List Window Overview (with symbolic values).



```
      ns        /top/clk        /top/pdata        /top/sdata
       delta      /top/prw          /top/srw
                  /top/pstrb        /top/sstrb
                     /top/prdy         /top/srdy
                        /top/paddr        /top/saddr

     900  +0          1 0 1 1  4       4 0 0 1  4       z
     905  +0          1 0 1 1  4       4 0 1 1  4       4
     920  +0          0 0 1 1  4       4 0 1 1  4       4
     940  +0          1 0 1 1  4       4 0 1 1  4       4
     960  +0          0 0 1 1  4       4 0 1 1  4       4
     980  +0          1 0 1 1  4       4 0 1 1  4       4
     985  +0          1 0 1 1  4       4 0 1 0  4       4
     990  +0          1 0 1 0  4       4 0 1 0  4       4
```

Aside from the List Signal Properties dialog, there are three other ways to change the radix:

- Change the default radix for the current simulation using **Simulate** > **Runtime Options** (Main window)
- Change the default radix for the current simulation using the radix command.
- Change the default radix permanently by editing the DefaultRadix variable in the *modelsim.ini* file.

Saving the Window Format

By default all Wave and List window information is forgotten once you close the windows. If you want to restore the windows to a previously configured layout, you must save a window format file. Follow these steps:

1. Add the objects you want to the Wave or List window.
2. Edit and format the objects to create the view you want.
3. Save the format to a file by selecting **File** > **Save** > **Format**.

To use the format file, start with a blank Wave or List window and run the DO file in one of two ways:

- Invoke the do command from the command line:

```
VSIM> do <my_format_file>
```

- Select **File** > **Load**.

Note

Window format files are design-specific. Use them only with the design you were simulating when they were created.

Printing and Saving Waveforms in the Wave window

You can print the waveform display or save it as an encapsulated postscript (EPS) file.

Saving a .eps Waveform File and Printing in UNIX

Select **File** > **Print Postscript** (Wave window) to print all or part of the waveform in the current Wave window in UNIX, or save the waveform as a .eps file on any platform (see also the write wave command).

Printing from the Wave Window on Windows Platforms

Select **File** > **Print** (Wave window) to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers).

Printer Page Setup

Select **File** > **Page setup** or click the Setup button in the Write Postscript or Print dialog box to define how the printed page will appear.

Saving List Window Data to a File

Select **File** > **Write List** in the List window to save the data in one of these formats:

- **Tabular** — writes a text file that looks like the window listing

```
ns    delta      /a       /b      /cin     /sum    /cout
0       +0        X        X        U        X        U
0       +1        0        1        0        X        U
2       +0        0        1        0        X        U
```

- **Events** — writes a text file containing transitions during simulation

```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI** — writes a file in standard TSSI format; see also, the write tssi command.

```
0 00000000000000010?????????
2 00000000000000010???????1?
3 00000000000000010??????010
4 00000000000000010000000010
100 00000001000000010000000010
```

You can also save List window output using the write list command.

Combining Objects into Buses

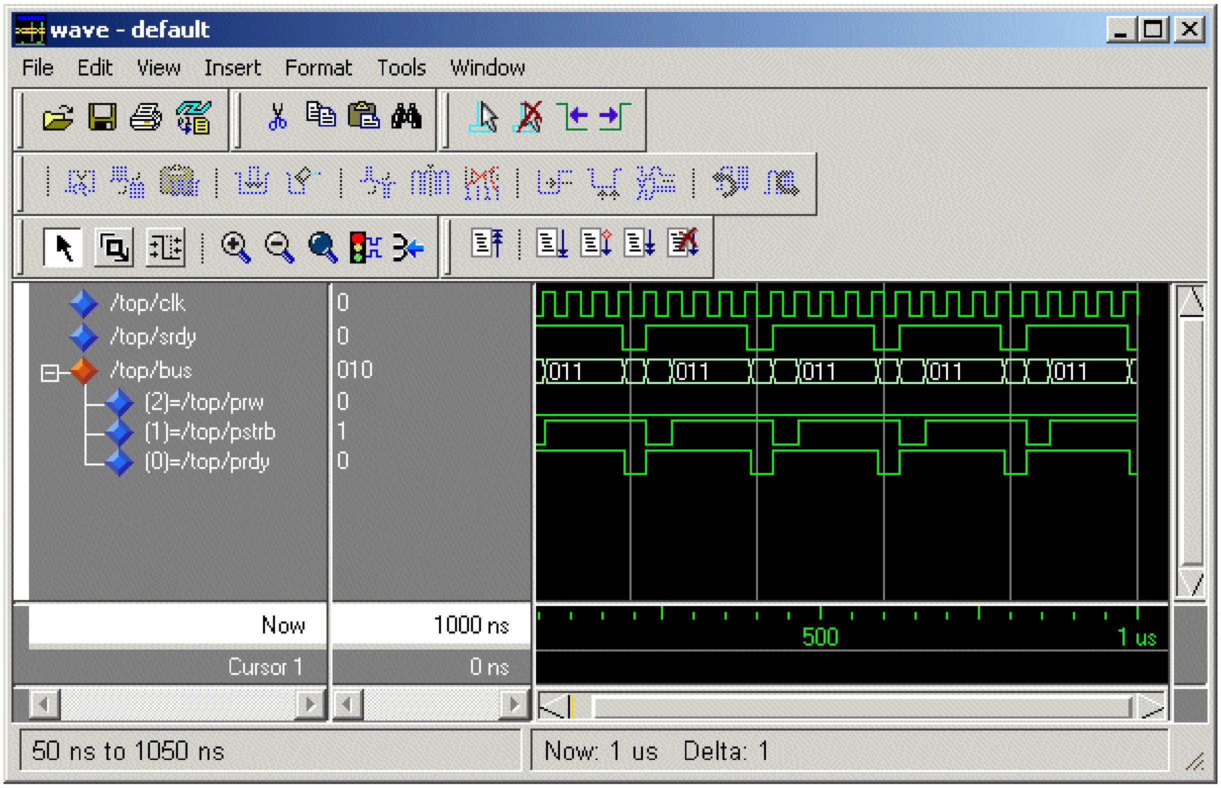
You can combine signals in the Wave or List window into buses. A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value. A virtual compare signal (the result of a comparison simulation) is not supported for combination with any other signal.

To combine signals into a bus, use one of the following methods:

- Select two or more signals in the Wave or List window and then choose **Tools > Combine Signals** from the menu bar. A virtual signal that is the result of a comparison simulation is not supported for combining with any other signal.
- Use the virtual signal command at the Main window command prompt.

Example

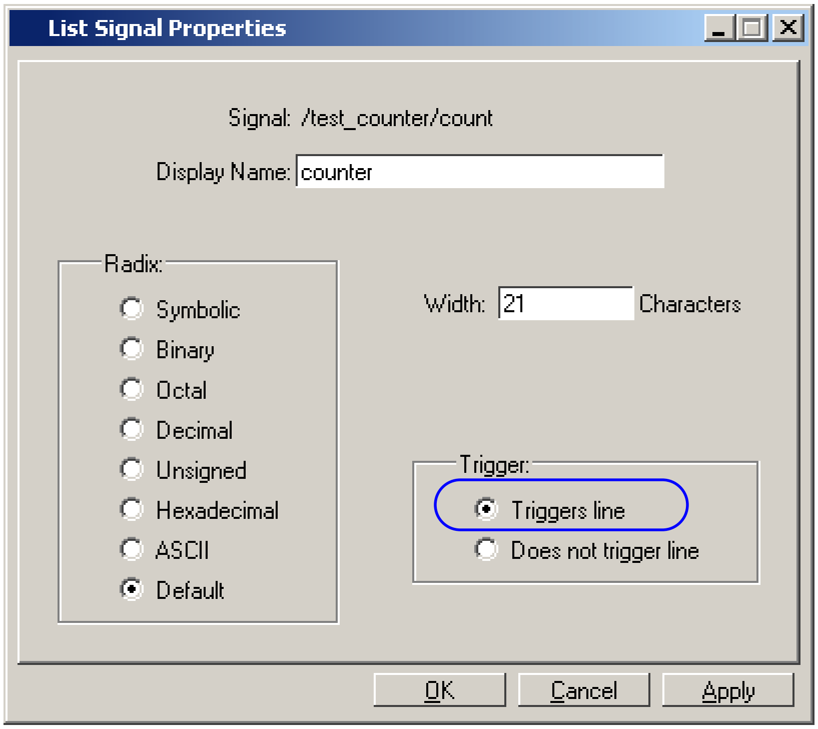
In the illustration below, three signals have been combined to form a new bus called "bus". Note that the component signals are listed in the order in which they were selected in the Wave window. Also note that the value of the bus is made up of the values of its component signals, arranged in a specific order.



Configuring New Line Triggering in the List Window

New line triggering refers to what events cause a new line of data to be added to the List window. By default ModelSim adds a new line for any signal change including deltas within a single unit of time resolution.

You can set new line triggering on a signal-by-signal basis or for the whole simulation. To set for a single signal, select **View** > **Signal Properties** from the List window menu bar (when the window is undocked) and select the **Triggers line** setting. Individual signal settings override global settings.



List Signal Properties

Signal: /test_counter/count

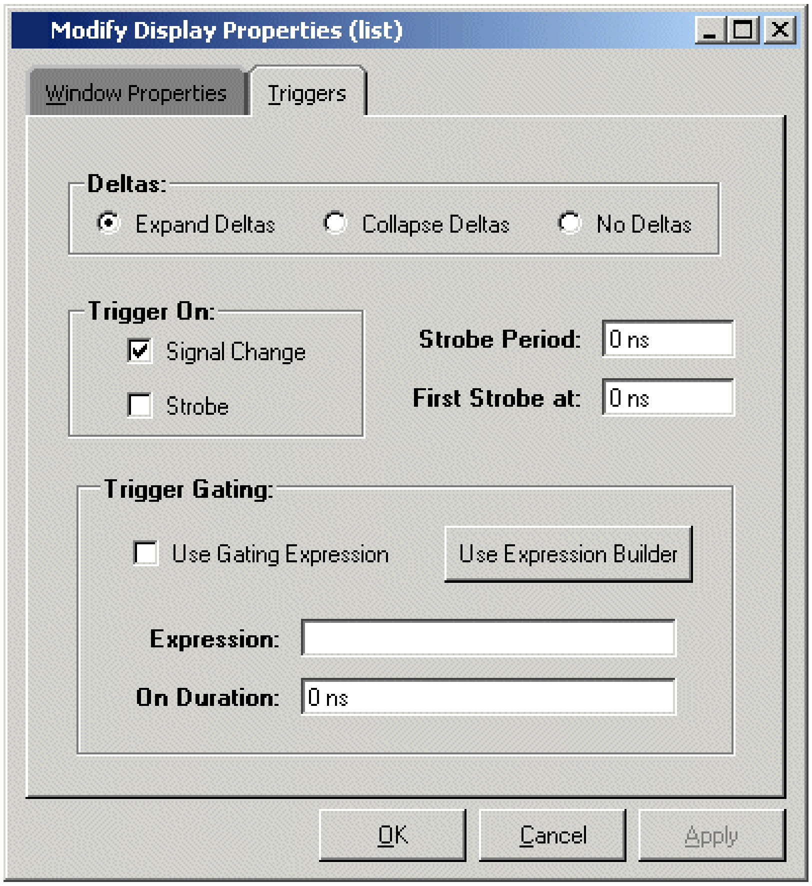
Display Name: counter

Radix:
○ Symbolic
○ Binary
○ Octal
○ Decimal
○ Unsigned
○ Hexadecimal
○ ASCII
● Default

Width: 21 Characters

Trigger:
● Triggers line
○ Does not trigger line

To modify new line triggering for the whole simulation, select **Tools** > **Window Preferences** from the List window menu bar (when the window is undocked), or use the configure command. When you select **Tools** > **Window Preferences**, the Modify Display Properties dialog appears:



Deltas:
● Expand Deltas ○ Collapse Deltas ○ No Deltas

Trigger On:
☑ Signal Change
□ Strobe

Strobe Period: 0 ns
First Strobe at: 0 ns

Trigger Gating:
□ Use Gating Expression
Expression:
On Duration: 0 ns

The following table summaries the options:

Table 12-5.

Option	Description
Deltas	Choose between displaying all deltas (Expand Deltas), displaying the value at the final delta (Collapse Delta). You can also hide the delta column all together (No Delta), however this will display the value at the final delta.
Strobe trigger	Specify an interval at which you want to trigger data display
Trigger gating	Use a gating expression to control triggering; see Using Gating Expressions to Control Triggering for more details

Using Gating Expressions to Control Triggering

Trigger gating controls the display of data based on an expression. Triggering is enabled once the gating expression evaluates to true. This setup behaves much like a hardware signal analyzer that starts recording data on a specified setup of address bits and clock edges.

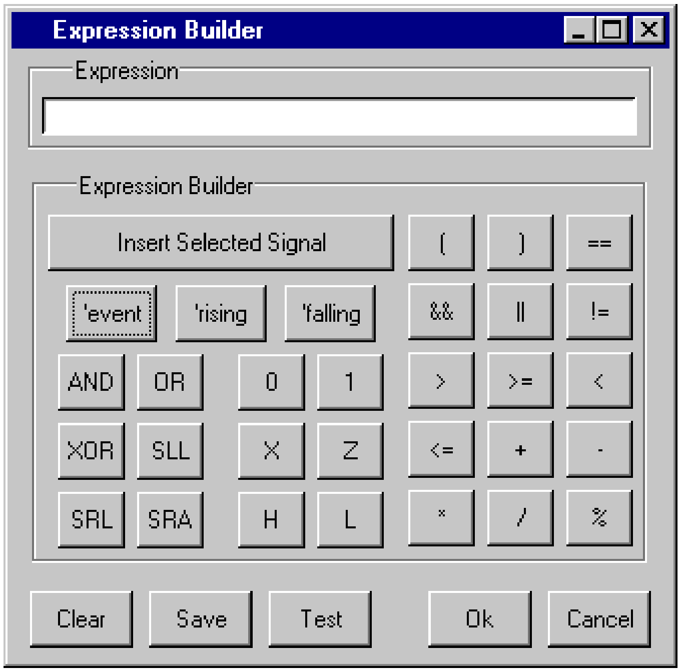
Here are some points about gating expressions:

- Gating expressions affect the display of data but not acquisition of the data.
- The expression is evaluated when the List window would normally have displayed a row of data (given the other trigger settings).
- The duration determines for how long triggering stays enabled after the gating expression returns to false (0). The default of 0 duration will enable triggering only while the expression is true (1). The duration is expressed in x number of default timescale units.
- Gating is level-sensitive rather than edge-triggered.

Trigger Gating Example Using the Expression Builder

This example shows how to create a gating expression with the ModelSim Expression Builder. Here is the procedure:

1. Select **Tools > Window Preferences** from the List window menu bar (when the window is undocked) and select the Triggers tab.
2. Click the **Use Expression Builder** button.



3. Select the signal in the List window that you want to be the enable signal by clicking on its name in the header area of the List window.
4. Click **Insert Selected Signal** and then **'rising** in the Expression Builder.
5. Click OK to close the Expression Builder.

   You should see the name of the signal plus "'rising" added to the Expression entry box of the Modify Display Properties dialog box.
6. Click **OK** to close the dialog.

If you already have simulation data in the List window, the display should immediately switch to showing only those cycles for which the gating signal is rising. If that isn't quite what you want, you can go back to the expression builder and play with it until you get it the way you want it.

If you want the enable signal to work like a "One-Shot" that would display all values for the next, say 10 ns, after the rising edge of enable, then set the **On Duration** value to **10 ns**.

Trigger Gating Example Using Commands

The following commands show the gating portion of a trigger configuration statement:

```
configure list -usegating 1
configure list -gateduration 100
configure list -gateexpr {/test_delta/iom_dd'rising}
```

See the configure command for more details.

Sampling Signals at a Clock Change

You easily can sample signals at a clock change using the add list command with the **-notrigger** argument. The **-notrigger** argument disables triggering the display on the specified signals. For example:

```
add list clk -notrigger a b c
```

When you run the simulation, List window entries for *clk*, *a*, *b*, and *c* appear only when *clk* changes.

If you want to display on rising edges only, you have two options:

1. Turn off the List window triggering on the clock signal, and then define a repeating strobe for the List window.
2. Define a "gating expression" for the List window that requires the clock to be in a specified state. See above.

Miscellaneous Tasks

Examining Waveform Values

You can use your mouse to display a dialog that shows the value of a waveform at a particular time. You can do this two ways:

- Rest your mouse pointer on a waveform. After a short delay, a dialog will pop-up that displays the value for the time at which your mouse pointer is positioned. If you'd prefer that this popup not display, it can be toggled off in the display properties. See Setting Wave Window Display Preferences.
- Right-click a waveform and select **Examine**. A dialog displays the value for the time at which you clicked your mouse. This method works in the List window as well.

Displaying Drivers of the Selected Waveform

You can automatically display in the Dataflow window the drivers of a signal selected in the Wave window. You can do this three ways:

- Select a waveform and click the Show Drivers button on the toolbar.
- Select a waveform and select Show Drivers from the shortcut menu
- Double-click a waveform edge (you can enable/disable this option in the display properties dialog; see Setting Wave Window Display Preferences)

This operation opens the Dataflow window and displays the drivers of the signal selected in the Wave window. The Wave pane in the Dataflow window also opens to show the selected signal with a cursor at the selected time. The Dataflow window shows the signal(s) values at the current cursor position.

Sorting a Group of Objects in the Wave Window

Select **View** > **Sort** to sort the objects in the pathname and values panes.

Creating and managing breakpoints

ModelSim supports both signal (i.e., when conditions) and file-line breakpoints. Breakpoints can be set from multiple locations in the GUI or from the command line.

Note

When running in full optimization mode, breakpoints may not be set. Run the design in non-optimized mode (or set +acc arguments) to enable you to set breakpoints in the design. See Restoring Full Design Visibility by Disabling Auto vopt and Design Object Visibility and the +acc Argument.

Breakpoints within SystemC portions of the design can only be set using File-line breakpoints.

Signal breakpoints

Signal breakpoints (when conditions) instruct ModelSim to perform actions when the specified conditions are met. For example, you can break on a signal value or at a specific simulator time (see the when command for additional details). When a breakpoint is hit, a message in the Main window transcript identifies the signal that caused the breakpoint.

Setting signal breakpoints from the command line

You use the when command to set a signal breakpoint from the VSIM> prompt.

Setting signal breakpoints from the GUI

Signal breakpoints are most easily set in the Objects Pane and the Wave Window Overview. Right-click a signal and select **Insert Breakpoint** from the context menu. A breakpoint is set on that signal and will be listed in the **Breakpoints** dialog.

File-line breakpoints

File-line breakpoints are set on executable lines in your source files. When the line is hit, the simulator stops and the Source window opens to show the line with the breakpoint. You can change this behavior by editing the PrefSource(OpenOnBreak) variable. See Simulator GUI Preferences for details on setting preference variables.

Since C Debug is invoked when you set a breakpoint within a SystemC module, your C Debug settings must be in place prior to setting a breakpoint. See Setting Up C Debug for more information. Once invoked, C Debug can be exited using the C Debug menu.

Setting file-line breakpoints from the command line

You use the bp command to set a file-line breakpoint from the VSIM> prompt.

Setting file-line breakpoints from the GUI

File-line breakpoints are most easily set using your mouse in the Source Window. Click on a blue line number at the left side of the Source window, and a red diamond denoting a breakpoint will appear. The breakpoints are toggles – click once to create the colored diamond; click again to disable or enable the breakpoint. To delete the breakpoint completely, click the red diamond with your right mouse button, and select **Remove Breakpoint**.

Waveform Compare

The ModelSim Waveform Compare feature allows you to compare simulation runs. Differences encountered in the comparison are summarized and listed in the Main window transcript and are shown in the Wave and List windows. In addition, you can write a list of the differences to a file using the compare info command.

Note

The Waveform Compare feature is available as an add-on to the PE or LE versions. Contact Mentor Graphics sales for more information.

The basic steps for running a comparison are as follows:

1. Run one simulation and save the dataset. For more information on saving datasets, see Saving a Simulation to a WLF File.
2. Run a second simulation.
3. Setup and run a comparison.
4. Analyze the differences in the Wave or List window.

Mixed-Language Waveform Compare Support

Mixed-language compares are supported as listed in the following table:

Table 12-6.

Language	Compares
C/C++ types	bool, char, unsigned char short, unsigned short int, unsigned int long, unsigned long
SystemC types	sc_bit, sc_bv, sc_logic, sc_lv sc_int, sc_uint sc_bigint, sc_biguint sc_signed, sc_unsigned

Table 12-6.

Language	Compares
Verilog types	net, reg

The number of elements must match for vectors; specific indexes are ignored.

Three Options for Setting up a Comparison

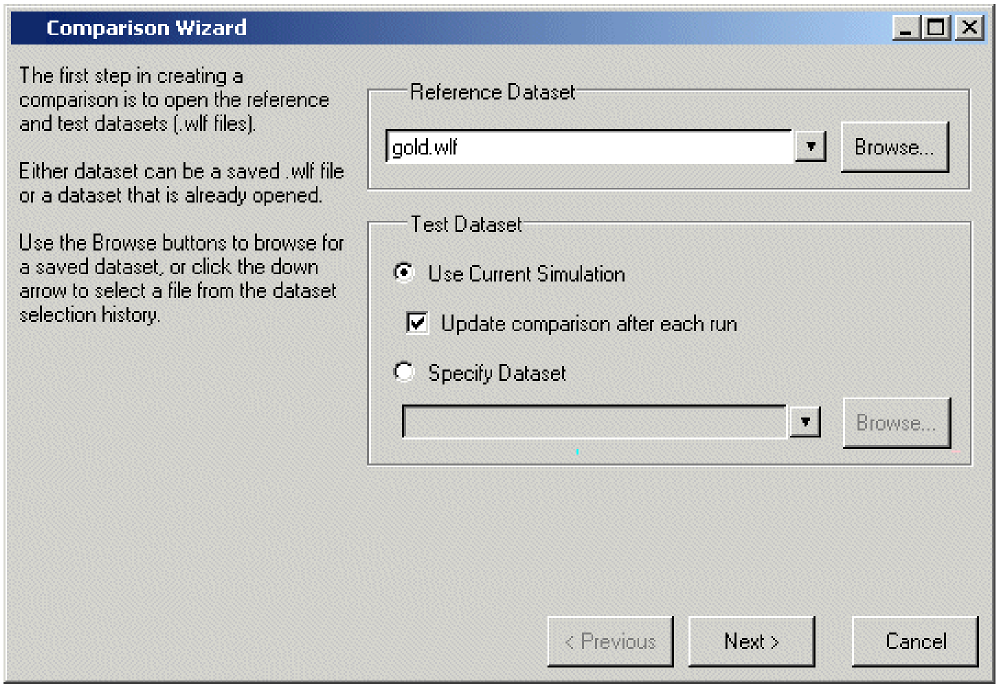
There are three options for setting up a comparison:

- Comparison Wizard – A series of dialogs that "walk" you through the process
- Comparison commands – Use a series of **compare** commands
- GUI – Use various dialogs to "manually" configure the comparison

Comparison Wizard

The simplest method for setting up a comparison is using the Wizard. The wizard is a series of dialogs that walks you through the process. To start the Wizard, select **Tools > Waveform Compare > Comparison Wizard** from either the Wave or Main window.

The graphic below shows the first dialog in the Wizard. As you can see from this example, the dialogs include instructions on the left-hand side.



Comparison Wizard

The first step in creating a comparison is to open the reference and test datasets (.wlf files).

Either dataset can be a saved .wlf file or a dataset that is already opened.

Use the Browse buttons to browse for a saved dataset, or click the down arrow to select a file from the dataset selection history.

Reference Dataset: gold.wlf

Test Dataset

● Use Current Simulation

☑ Update comparison after each run

○ Specify Dataset

Comparison Graphic Interface

You can also set up a comparison via the GUI without using the Wizard. The steps of this process are described further in Setting Up a Comparison with the GUI.

Comparison Commands

There are numerous commands that give you complete control over a comparison. These commands can be entered in the Main window transcript or run via a DO file. The commands are detailed in the Reference Manual, but the following example shows the basic sequence:

```
compare start gold.wlf vsim.wlf
compare add /*
compare run
```

Comparing Signals with Different Names

You can use the compare add command to specify a comparison between two signals with different names.

Setting Up a Comparison with the GUI

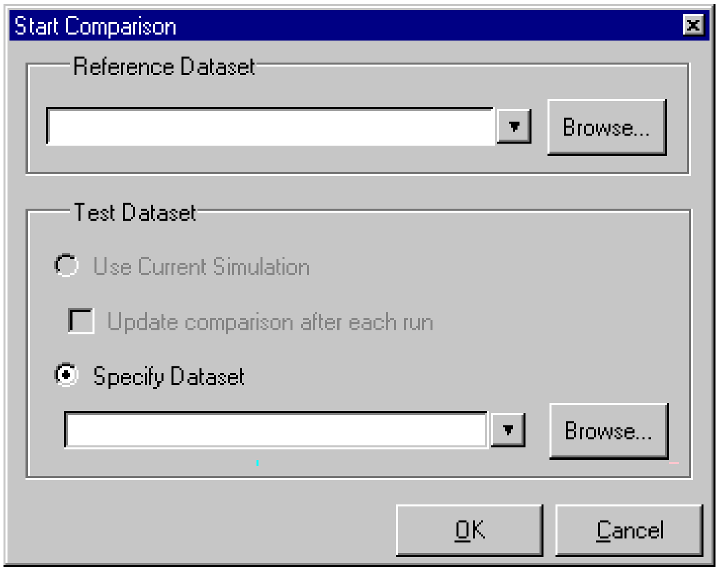
To setup a comparison with the GUI, follow these steps:

1. Initiate the comparison by specifying the reference and test datasets. See Starting a Waveform Comparison for details.
2. Add objects to the comparison. See Adding Signals, Regions, and Clocks for details.
3. Specify the comparison method. See Specifying the Comparison Method for details.
4. Configure comparison options. See Setting Compare Options for details.
5. Run the comparison by selecting Tools > Waveform Compare > Run Comparison.
6. View the results. See Viewing Differences in the Wave Window, Viewing Differences in the List Window, and Viewing Differences in Textual Formatfor details.

Waveform Compare is initiated from either the Main or Wave window by selecting **Tools >Waveform Compare > Start Comparison**.

Starting a Waveform Comparison

Select **Tools >Waveform Compare > Start Comparison** to initiate the comparison. The Start Comparison dialog box allows you define the Reference and Test datasets.



○ Use Current Simulation

☐ Update comparison after each run

● Specify Dataset

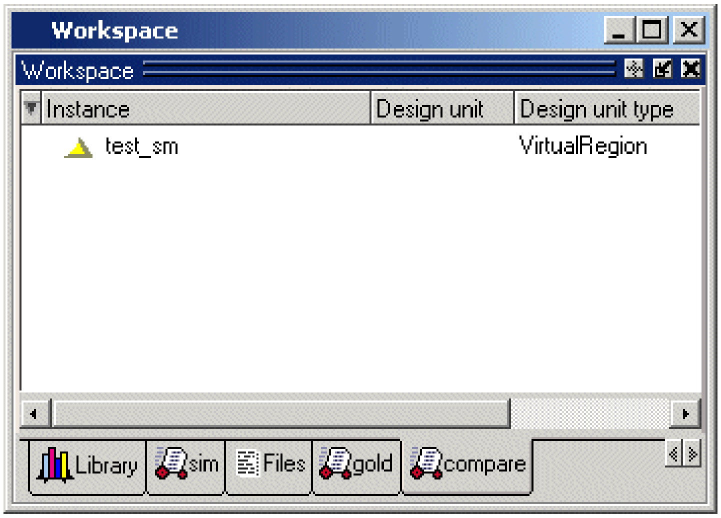
Reference Dataset

The Reference Dataset is the .wlf file to which the test dataset will be compared. It can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

Test Dataset

The Test Dataset is the .wlf file that will be compared against the Reference Dataset. Like the Reference Dataset, it can be a saved dataset, the current simulation dataset, or any part of the current simulation dataset.

Once you click **OK** in the Start Comparison dialog box, ModelSim adds a Compare tab to the Main window.



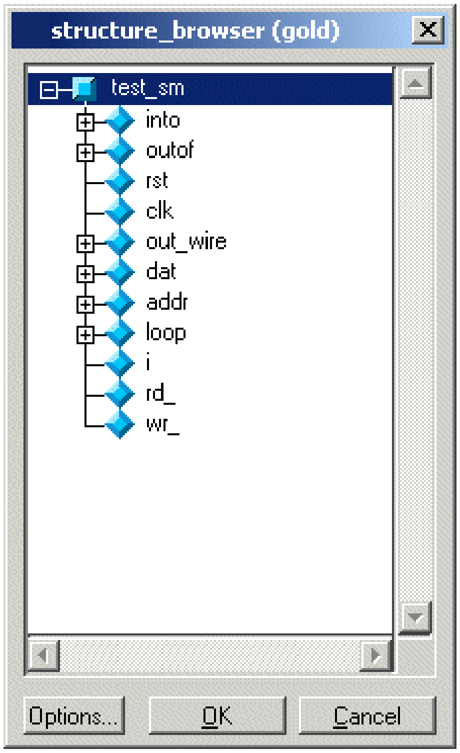
After adding the signals, regions, and/or clocks you want to use in the comparison (see Adding Signals, Regions, and Clocks), you will be able to drag compare objects from this tab into the Wave and List windows.

Adding Signals, Regions, and Clocks

To designate the signals, regions, or clocks to be used in the comparison, click **Tools > Waveform Compare > Add**.

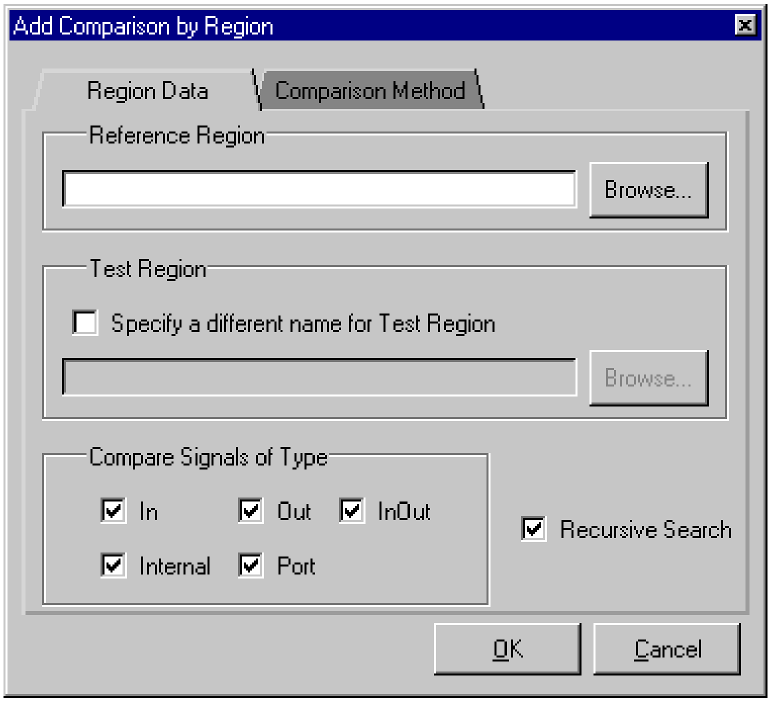
Adding Signals

Clicking **Tools** > **Waveform Compare** > **Add** > **Compare by Signal** in the Wave window opens the structure_browser window, where you can specify signals to be used in the comparison.



Adding Regions

Rather than comparing individual signals, you can also compare entire regions of your design. Select **Tools > Waveform Compare > Add > Compare by Region** to open the Add Comparison by Region dialog.



☐ Specify a different name for Test Region

☑ In ☑ Out ☑ InOut

☑ Internal ☑ Port

☑ Recursive Search

Adding Clocks

You add clocks when you want to perform a clocked comparison. See Specifying the Comparison Method for details.

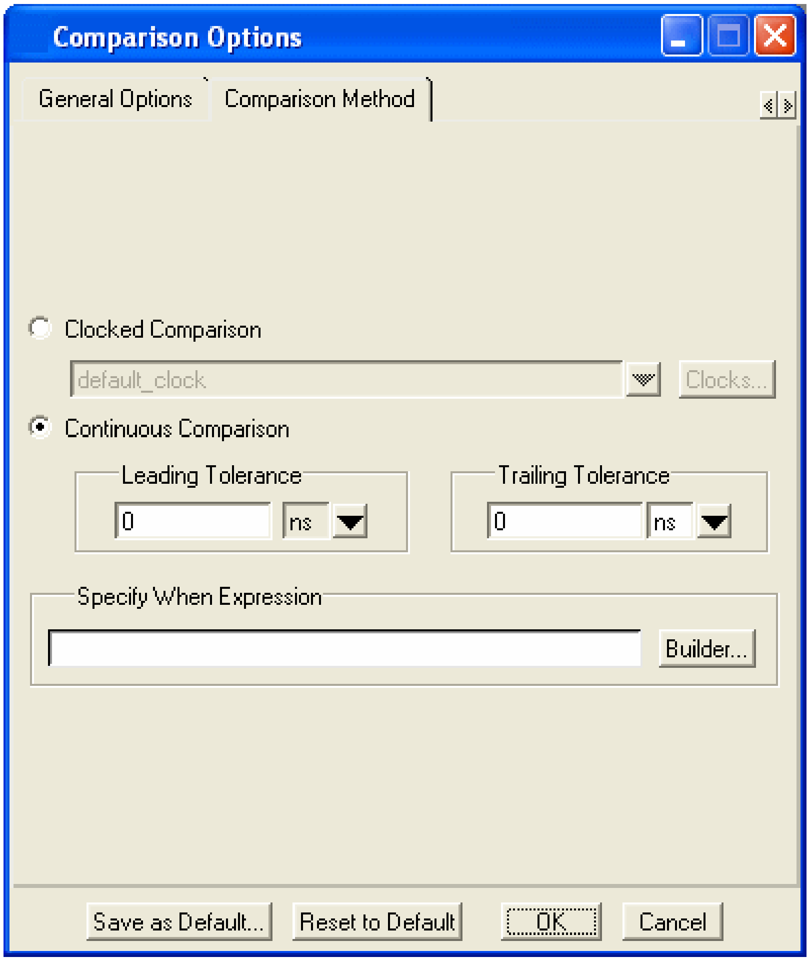
Specifying the Comparison Method

The Waveform Compare feature provides two comparison methods:

- Continuous comparison — Test signals are compared to reference signals at each transition of the reference. Timing differences between the test and reference signals are shown with rectangular red markers in the Wave window and yellow markers in the List window.
- Clocked comparisons — Signals are compared only at or just after an edge on some signal. In this mode, you define one or more clocks. The test signal is compared to a reference signal and both are sampled relative to the defined clock. The clock can be defined as the rising or falling edge (or either edge) of a particular signal plus a user-specified delay. The design need not have any events occurring at the specified clock

time. Differences between test signals and the clock are highlighted with red diamonds in the Wave window.

To specify the comparison method, select **Tools > Waveform Compare > Options** and select the Comparison Method tab.



Comparison Options

○ Clocked Comparison

default_clock

● Continuous Comparison

Leading Tolerance: 0 ns

Trailing Tolerance: 0 ns

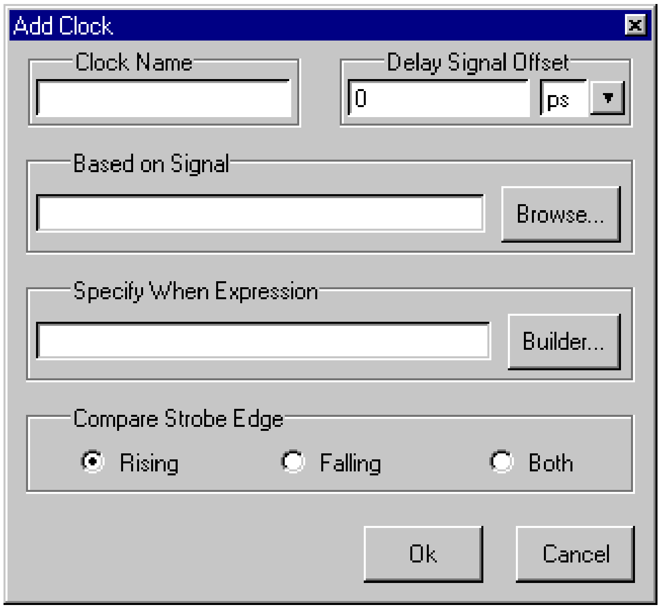
Specify When Expression:

Continuous Comparison

Continuous comparisons are the default. You have the option of specifying leading and trailing tolerances and a when expression that must evaluate to "true" or 1 at the signal edge for the comparison to become effective.

Clocked Comparison

To specify a clocked comparison you must define a clock in the Add Clock dialog. You can access this dialog via the Clocks button in the Comparison Method tab or by selecting **Tools** > **Waveform Compare** > **Add** > **Clocks**.

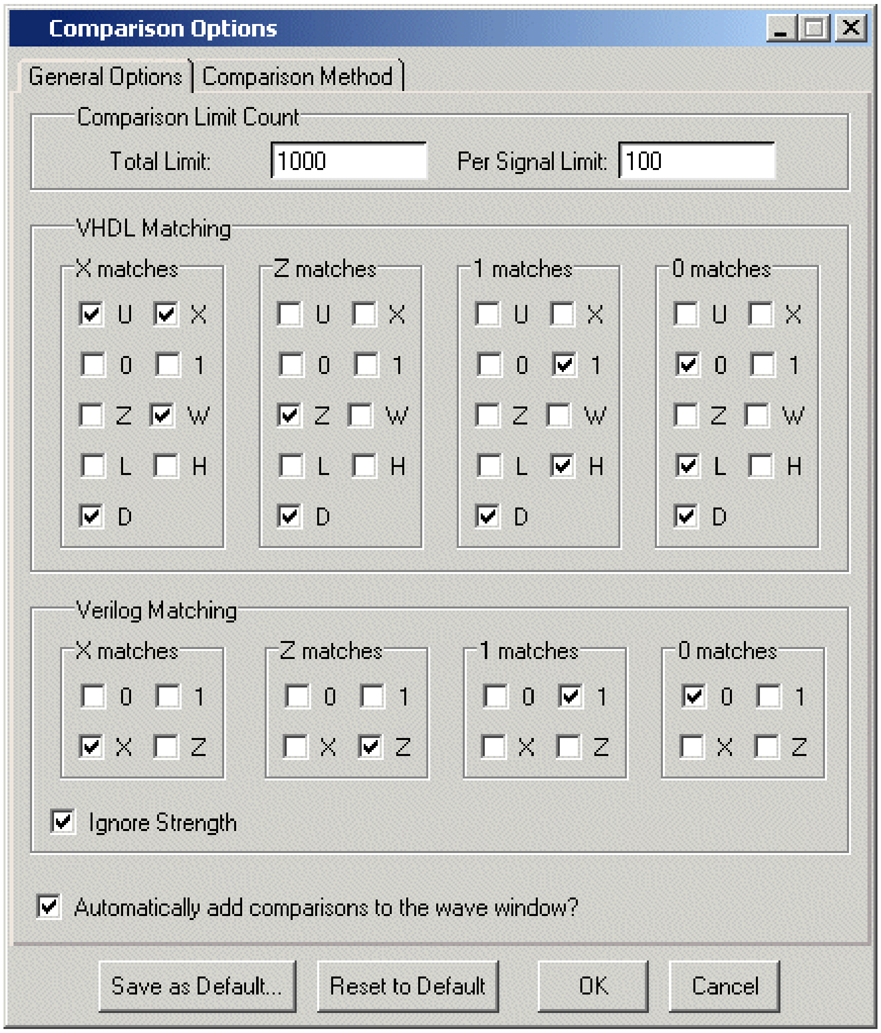


Delay Signal Offset: 0 ps

Compare Strobe Edge: ● Rising ○ Falling ○ Both

Setting Compare Options

There are a few "global" options that you can set for a comparison. Select **Tools > Waveform Compare > Options**.



Comparison Limit Count

Total Limit: 1000   Per Signal Limit: 100

VHDL Matching

X matches: ☑ U ☑ X □ 0 □ 1 □ Z ☑ W □ L □ H ☑ D

Z matches: □ U □ X □ 0 □ 1 ☑ Z □ W □ L □ H ☑ D

1 matches: □ U □ X □ 0 ☑ 1 □ Z □ W □ L ☑ H ☑ D

0 matches: □ U □ X ☑ 0 □ 1 □ Z □ W ☑ L □ H ☑ D

Verilog Matching

X matches: □ 0 □ 1 ☑ X □ Z

Z matches: □ 0 □ 1 □ X ☑ Z

1 matches: □ 0 ☑ 1 □ X □ Z

0 matches: ☑ 0 □ 1 □ X □ Z

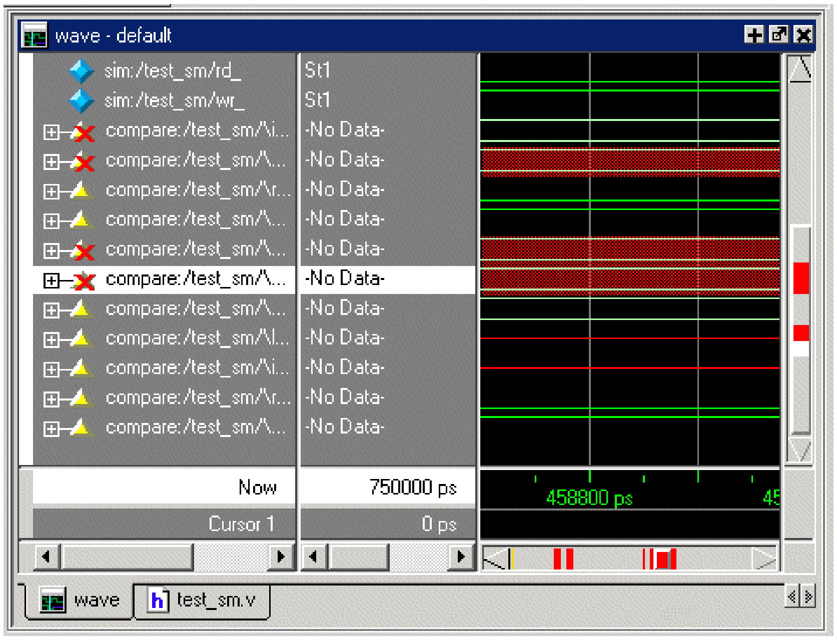
☑ Ignore Strength

☑ Automatically add comparisons to the wave window?

Options in this dialog include setting the maximum number of differences allowed before the comparison terminates, specifying signal value matching rules, and saving or resetting the defaults.

Viewing Differences in the Wave Window

The Wave window provides a graphic display of comparison results. Pathnames of all test signals included in the comparison are denoted by yellow triangles. Test signals that contain timing differences when compared with the reference signals are denoted by a red X over the yellow triangle.



wave - default

sim:/test_sm/rd_	St1
sim:/test_sm/wr_	St1
compare:/test_sm/\i...	-No Data-
compare:/test_sm/\...	-No Data-
compare:/test_sm/\r...	-No Data-
compare:/test_sm/\...	-No Data-
compare:/test_sm/\...	-No Data-
compare:/test_sm/\...	-No Data-
compare:/test_sm/\...	-No Data-
compare:/test_sm/\l...	-No Data-
compare:/test_sm/\i...	-No Data-
compare:/test_sm/\r...	-No Data-
compare:/test_sm/\...	-No Data-
Now	750000 ps
Cursor 1	0 ps

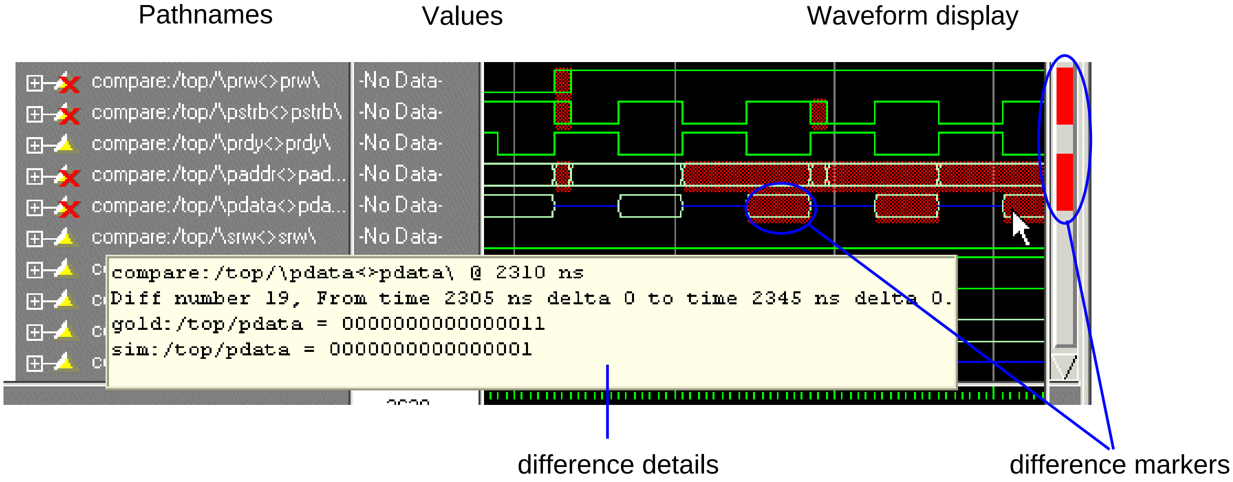
The names of the comparison objects take the form:

```
<path>/\refSignalName<>testSignalName\
```

If you compare two signals from different regions, the signal names include the uncommon part of the path.

In comparisons of signals with multiple bits, you can display them in "buswise" or "bitwise" format. Buswise format lists the busses under the compare object whereas bitwise format lists each individual bit under the compare object. To select one format or the other, click your right mouse button on the plus sign (’+’) next to a compare object.

Timing differences are also indicated by red bars in the vertical and horizontal scroll bars of the waveform display, and by red difference markers on the waveforms themselves. Rectangular difference markers denote continuous differences. Diamond difference markers denote clocked differences. Placing your mouse cursor over any difference marker will initiate a popup display that provides timing details for that difference.



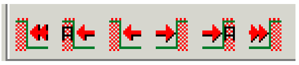
```
compare:/top/\pdata<>pdata\ @ 2310 ns
Diff number 19, From time 2305 ns delta 0 to time 2345 ns delta 0.
gold:/top/pdata = 0000000000000011
sim:/top/pdata = 0000000000000001
```

The values column of the Wave window displays the words "match" or "diff" for every test signal, depending on the location of the selected cursor. "Match" indicates that the value of the test signal matches the value of the reference signal at the time of the selected cursor. "Diff" indicates a difference between the test and reference signal values at the selected cursor.

Annotating Differences

You can tag differences with textual notes that are included in the difference details popup and comparison reports. Click a difference with the right mouse button, and select **Annotate Diff**. Or, use the compare annotate command.

Compare Icons



The Wave window includes six comparison icons that let you quickly jump between differences. From left to right, the icons do the following: find first difference, find previous annotated difference, find previous difference, find next difference, find next annotated difference, find last difference. Use these icons to move the selected cursor.

These buttons cycle through differences on all signals. To view differences for just the selected signal, press <tab> and <shift - tab> on your keyboard.

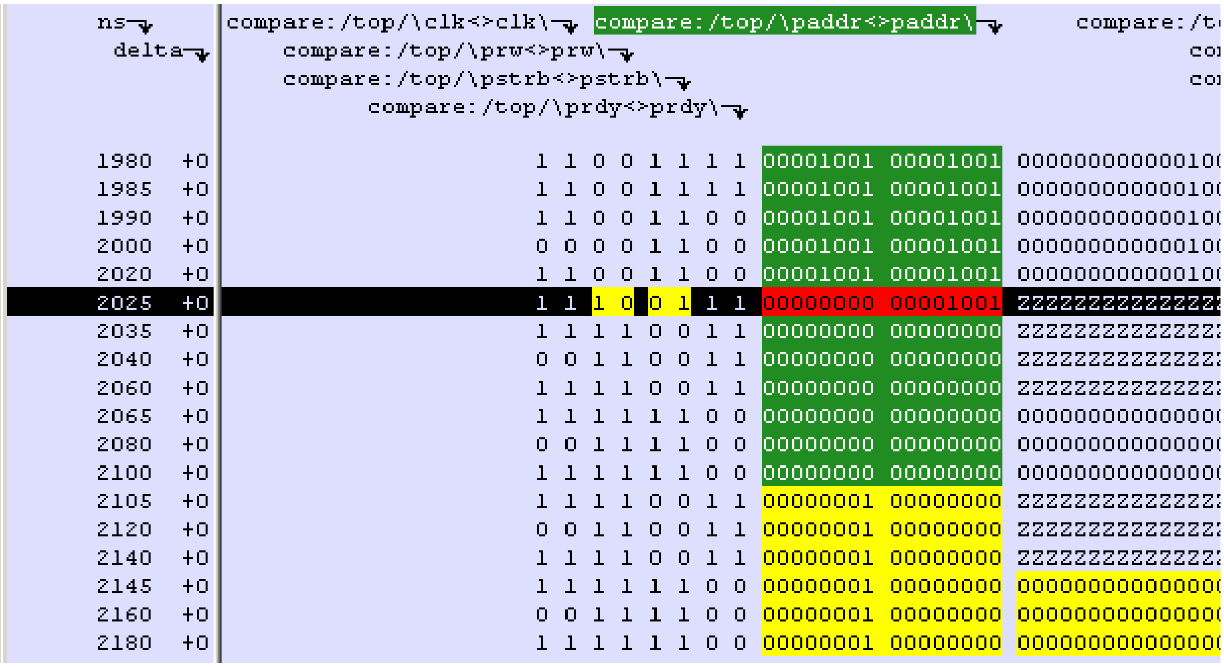
Note

If you have differences on individual bits of a bus, the compare icons will stop on those differences but <tab> and <shift - tab> will not.

The compare icons cycle through comparison objects in all open Wave windows. If you have two Wave windows displayed, each containing different comparison objects, the compare icons will cycle through the differences displayed in both windows.

Viewing Differences in the List Window

Compare objects can be displayed in the List window too. Differences are highlighted with a yellow background. Tabbing on selected columns moves the selection to the next difference (actually difference edge). Shift-tabbing moves the selection backwards.



```
      ns        compare:/top/\clk<>clk\   compare:/top/\paddr<>paddr\       compare:/t
       delta        compare:/top/\prw<>prw\                                        co
                    compare:/top/\pstrb<>pstrb\                                    co
                           compare:/top/\prdy<>prdy\

      1980  +0                          1 1 0 0 1 1 1 1 00001001 00001001 000000000000010
      1985  +0                          1 1 0 0 1 1 1 1 00001001 00001001 000000000000010
      1990  +0                          1 1 0 0 1 1 0 0 00001001 00001001 000000000000010
      2000  +0                          0 0 0 0 1 1 0 0 00001001 00001001 000000000000010
      2020  +0                          1 1 0 0 1 1 0 0 00001001 00001001 000000000000010
      2025  +0                          1 1 1 0 0 1 1 1 00000000 00001001 ZZZZZZZZZZZZZZ
      2035  +0                          1 1 1 1 0 0 1 1 00000000 00000000 ZZZZZZZZZZZZZZ
      2040  +0                          0 0 1 1 0 0 1 1 00000000 00000000 ZZZZZZZZZZZZZZ
      2060  +0                          1 1 1 1 0 0 1 1 00000000 00000000 ZZZZZZZZZZZZZZ
      2065  +0                          1 1 1 1 1 1 0 0 00000000 00000000 000000000000000
      2080  +0                          0 0 1 1 1 1 0 0 00000000 00000000 000000000000000
      2100  +0                          1 1 1 1 1 1 0 0 00000000 00000000 000000000000000
      2105  +0                          1 1 1 1 0 0 1 1 00000001 00000000 ZZZZZZZZZZZZZZ
      2120  +0                          0 0 1 1 0 0 1 1 00000001 00000000 ZZZZZZZZZZZZZZ
      2140  +0                          1 1 1 1 0 0 1 1 00000001 00000000 ZZZZZZZZZZZZZZ
      2145  +0                          1 1 1 1 1 1 0 0 00000001 00000000 000000000000000
      2160  +0                          0 0 1 1 1 1 0 0 00000001 00000000 000000000000000
      2180  +0                          1 1 1 1 1 1 0 0 00000001 00000000 000000000000000
```

Right-clicking on a yellow-highlighted difference gives you three options: **Diff Info**, **Annotate Diff**, and **Ignore/Noignore** diff. With these options you can elect to display difference information, you can ignore selected differences or turn off ignore, and you can annotate individual differences.

Viewing Differences in Textual Format

You can also view text output of the differences either in the Transcript pane of the Main window or in a saved file. To view them in the transcript, select **Tools > Waveform Compare > Differences > Show**. To save them to a text file, select **Tools > Waveform Compare > Differences > Write Report**.

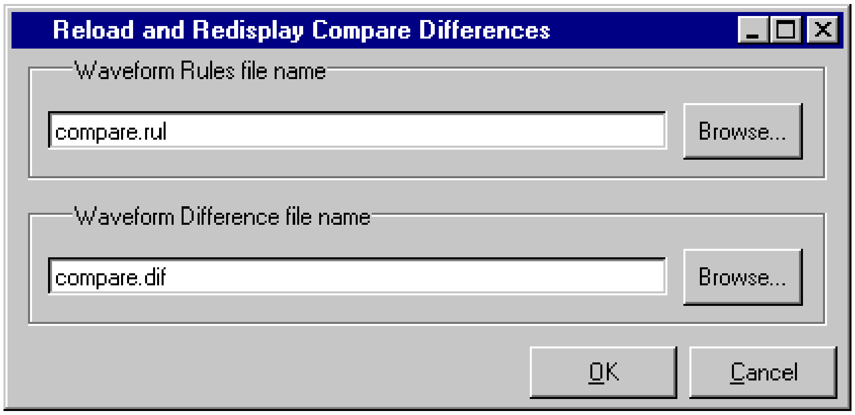
Saving and Reloading Comparison Results

To save comparison results for future use, you must save both the comparison setup rules and the comparison differences.

To save the rules, select **Tools** > **Waveform Compare** > **Rules** > **Save**. This file will contain all rules for reproducing the comparison. The default file name is "compare.rul."

To save the differences, select **Tools** > **Waveform Compare** > **Differences** > **Save**. The default file name is "compare.dif".

To reload the comparison results at a later time, select **Tools** > **Waveform Compare** > **Reload** and specify the rules and difference files.



Waveform Rules file name: compare.rul

Waveform Difference file name: compare.dif

Comparing Hierarchical and Flattened Designs

If you are comparing a hierarchical RTL design simulation against a flattened synthesized design simulation, you may have different hierarchies, different signal names, and the buses may be broken down into one-bit signals in the gate-level design. All of these differences can be handled by ModelSim’s Waveform Compare feature.

- If the test design is hierarchical but the hierarchy is different from the hierarchy of the reference design, you can use the compare add command to specify which region path in the test design corresponds to that in the reference design.
- If the test design is flattened and test signal names are different from reference signal names, the compare add command allows you to specify which signal in the test design will be compared to which signal in the reference design.
- If, in addition, buses have been dismantled, or "bit-blasted", you can use the **-rebuild** option of the compare add command to automatically rebuild the bus in the test design. This will allow you to look at the differences as one bus versus another.

If signals in the RTL test design are different in type from the synthesized signals in the reference design – registers versus nets, for example – the Waveform Compare feature will automatically do the type conversion for you. If the type differences are too extreme (say integer versus real), Waveform Compare will let you know.

Chapter 13
Tracing Signals with the Dataflow Window

This chapter discusses how to use the Dataflow window for tracing signal values and browsing the physical connectivity of your design.

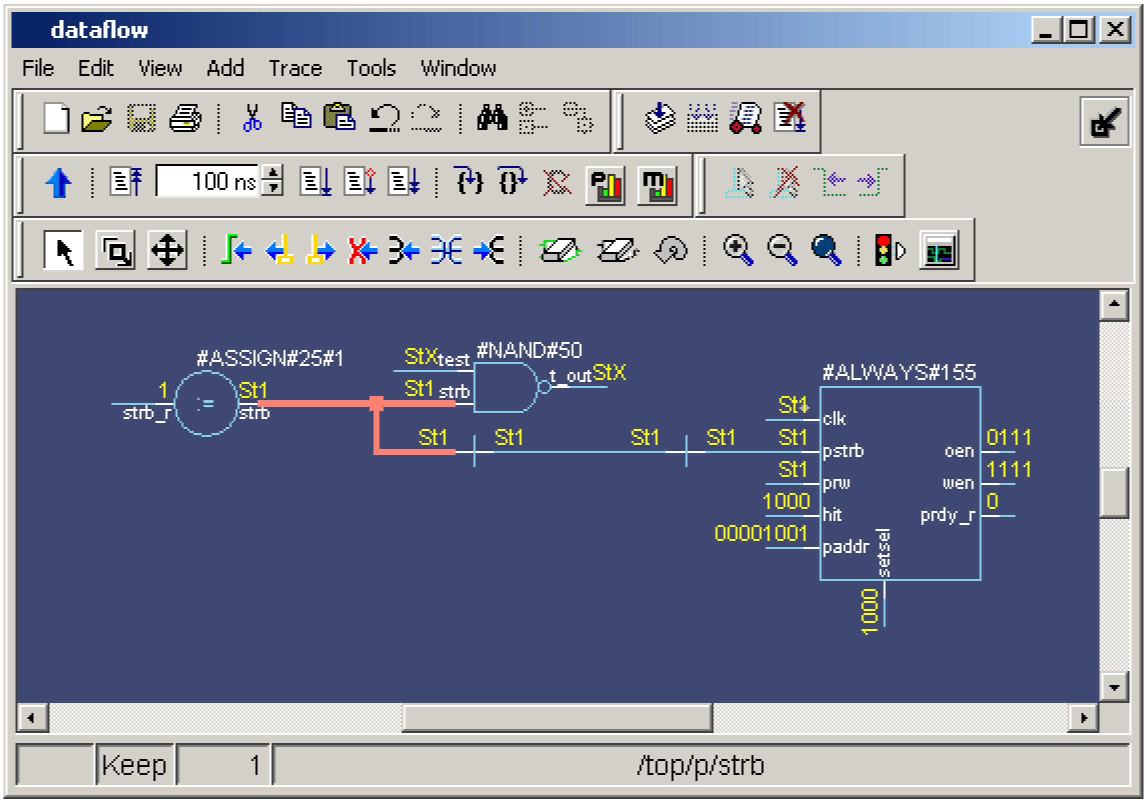
Dataflow Window Overview

The Dataflow window allows you to explore the "physical" connectivity of your design; to trace events that propagate through the design; and to identify the cause of unexpected outputs.

Note

ModelSim versions operating without a dataflow license feature have limited Dataflow functionality. Without the license feature, the window displays the message “Extended mode disabled” and will show only one process and its attached signals or one signal and its attached processes. Contact your Mentor Graphics sales representative if you currently don’t have a dataflow feature.

Figure 13-1. The Dataflow window (undocked)



Objects You Can View in the Dataflow Window

The Dataflow window displays:

- processes
- signals, nets, and registers
- and interconnect

The window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. See Symbol Mapping for details.

You cannot view SystemC objects in the Dataflow window; however, you can view HDL regions from mixed designs that include SystemC.

Adding Objects to the Window

You can use any of the following methods to add objects to the Dataflow window:

- drag and drop objects from other windows
- use the Navigate menu options in the Dataflow window
- use the add dataflow command
- double-click any waveform in the Wave window display

The **Navigate** menu offers four commands that will add objects to the window. The commands include:

- **View region** — clear the window and display all signals from the current region
- **Add region** — display all signals from the current region without first clearing window
- **View all nets** — clear the window and display all signals from the entire design
- **Add ports** — add port symbols to the port signals in the current region

When you view regions or entire nets, the window initially displays only the drivers of the added objects in order to reduce clutter. You can easily view readers by selecting an object and invoking **Navigate > Expand net to readers**.

A small circle above an input signal on a block denotes a trigger signal that is on the process' sensitivity list.

Links to Other Windows

The Dataflow window has links to other windows as described below:

Table 13-1.

Window	Link
Main Window	select a signal or process in the Dataflow window, and the structure tab updates if that object is in a different design unit
Active Processes Pane	select a process in either window, and that process is highlighted in the other
Objects Pane	select a design object in either window, and that object is highlighted in the other
Wave Window	• trace through the design in the Dataflow window, and the associated signals are added to the Wave window • move a cursor in the Wave window, and the values update in the Dataflow window
Source Window	select an object in the Dataflow window, and the Source window updates if that object is in a different source file

Exploring the Connectivity of the Design

A primary use of the Dataflow window is exploring the "physical" connectivity of your design. One way of doing this is by expanding the view from process to process. This allows you to see the drivers/receivers of a particular signal, net, or register.

You can expand the view of your design using menu commands or your mouse. To expand with the mouse, simply double click a signal, register, or process. Depending on the specific object you click, the view will expand to show the driving process and interconnect, the reading process and interconnect, or both.

Alternatively, you can select a signal, register, or net, and use one of the toolbar buttons or menu commands described below:

Expand net to all drivers display driver(s) of the selected signal, net, or register	Navigate > Expand net to drivers
Expand net to all drivers and readers display driver(s) and reader(s) of the selected signal, net, or register	Navigate > Expand net

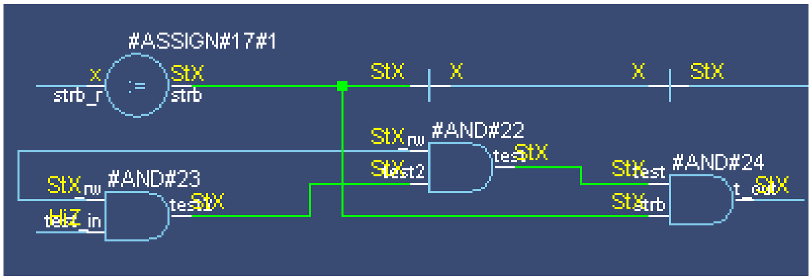
	Expand net to all readers display reader(s) of the selected signal, net, or register	Navigate > Expand net to readers

As you expand the view, note that the "layout" of the design may adjust to best show the connectivity. For example, the location of an input signal may shift from the bottom to the top of a process.

Tracking Your Path Through the Design

You can quickly traverse through many components in your design. To help mark your path, the objects that you have expanded are highlighted in green.

Figure 13-2. Green highlighting shows path you've taken through the design



You can clear this highlighting using the **Edit > Erase highlight** command or by clicking the Erase highlight icon in the toolbar.

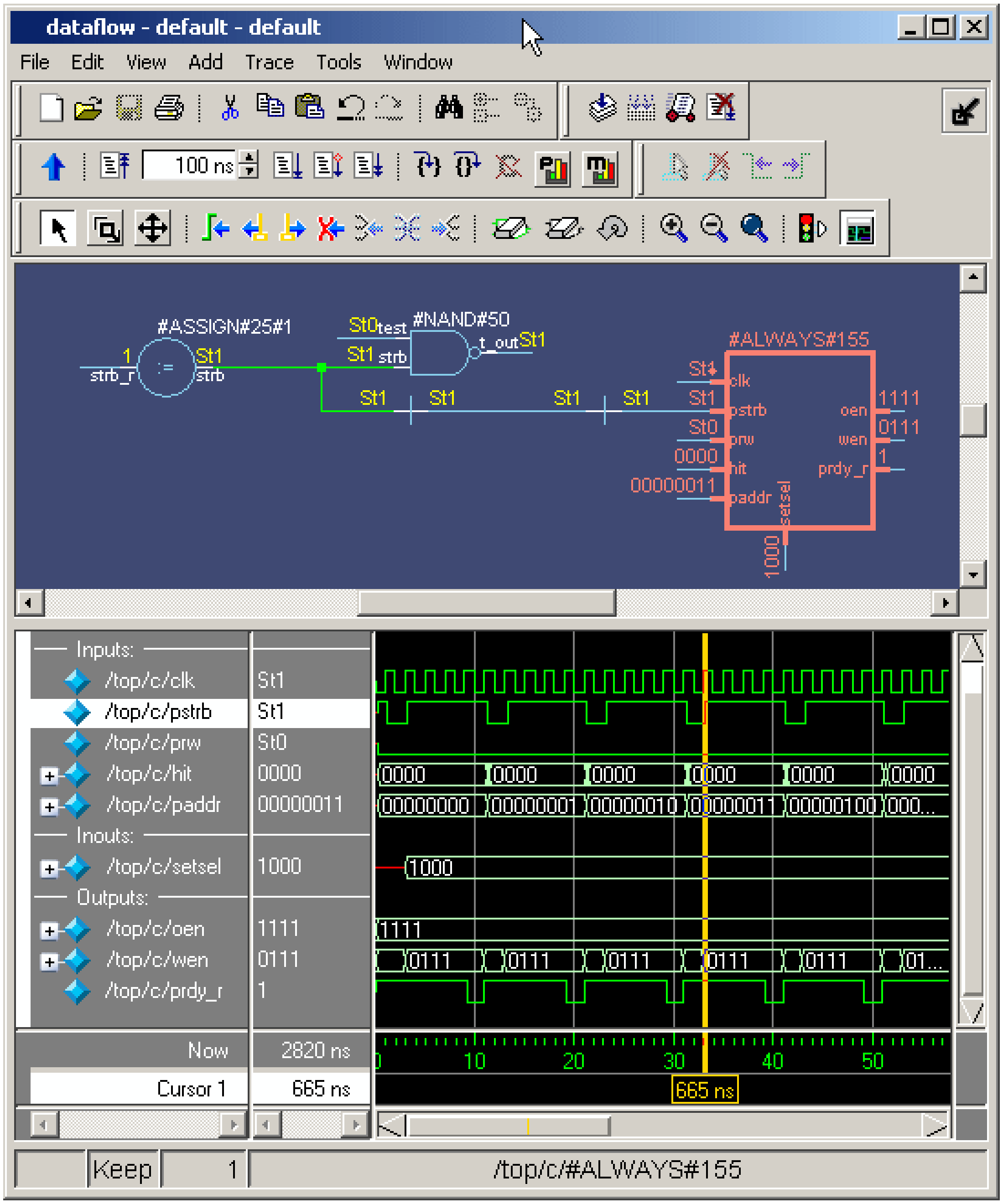
The Embedded Wave Viewer

Another way of exploring your design is to use the Dataflow window's embedded wave viewer. This viewer closely resembles, in appearance and operation, the stand-alone Wave window (see Waveform Analysis for more information).

The wave viewer is opened using the **View > Show Wave** command or by clicking the Show Wave icon.

One common scenario is to place signals in the wave viewer and the Dataflow panes, run the design for some amount of time, and then use time cursors to investigate value changes. In other words, as you place and move cursors in the wave viewer pane (see Measuring Time with Cursors in the Wave Window for details), the signal values update in the Dataflow pane.

Figure 13-3. Wave viewer displays inputs and outputs of selected process



/top/c/clk	St1
/top/c/pstrb	St1
/top/c/prw	St0
/top/c/hit	0000
/top/c/paddr	00000011
/top/c/setsel	1000
/top/c/oen	1111
/top/c/wen	0111
/top/c/prdy_r	1
Now	2820 ns
Cursor 1	665 ns

Another scenario is to select a process in the Dataflow pane, which automatically adds to the wave viewer pane all signals attached to the process.

See Tracing Events (Causality) for another example of using the embedded wave viewer.

Zooming and Panning

The Dataflow window offers several tools for zooming and panning the display.

These zoom buttons are available on the toolbar:

Zoom In
zoom in by a factor of two from the current view

Zoom Out
zoom out by a factor of two from current view

Zoom Full
zoom out to view the entire schematic

To zoom with the mouse, you can either use the middle mouse button or enter Zoom Mode by selecting **View** > **Zoom** and then use the left mouse button.

Four zoom options are possible by clicking and dragging in different directions:

- Down-Right: Zoom Area (In)
- Up-Right: Zoom Out (zoom amount is displayed at the mouse cursor)
- Down-Left: Zoom Selected
- Up-Left: Zoom Full

The zoom amount is displayed at the mouse cursor. A zoom operation must be more than 10 pixels to activate.

Panning with the Mouse

You can pan with the mouse in two ways: 1) enter Pan Mode by selecting **View** > **Pan** and then drag with the left mouse button to move the design; 2) hold down the <Ctrl> key and drag with the middle mouse button to move the design.

Tracing Events (Causality)

One of the most useful features of the Dataflow window is tracing an event to see the cause of an unexpected output. This feature uses the Dataflow window's embedded wave viewer (see The Embedded Wave Viewer for more details).

In short you identify an output of interest in the Dataflow pane and then use time cursors in the wave viewer pane to identify events that contribute to the output.

The process for tracing events is as follows:

1. Log all signals before starting the simulation (add log -r /*).
2. After running a simulation for some period of time, open the Dataflow window and the wave viewer pane.
3. Add a process or signal of interest into the Dataflow window (if adding a signal, find its driving process). Select the process and all signals attached to the selected process will appear in the wave viewer pane.

4. Place a time cursor on an edge of interest; the edge should be on a signal that is an output of the process.
5. Select **Trace > Trace input net to event**.

   A second cursor is added at the most recent input event.
6. Keep selecting **Trace** > **Trace next event** until you've reached an input event of interest. Note that the signals with the events are selected in the wave pane.
7. Now select **Trace** > **Trace Set**.

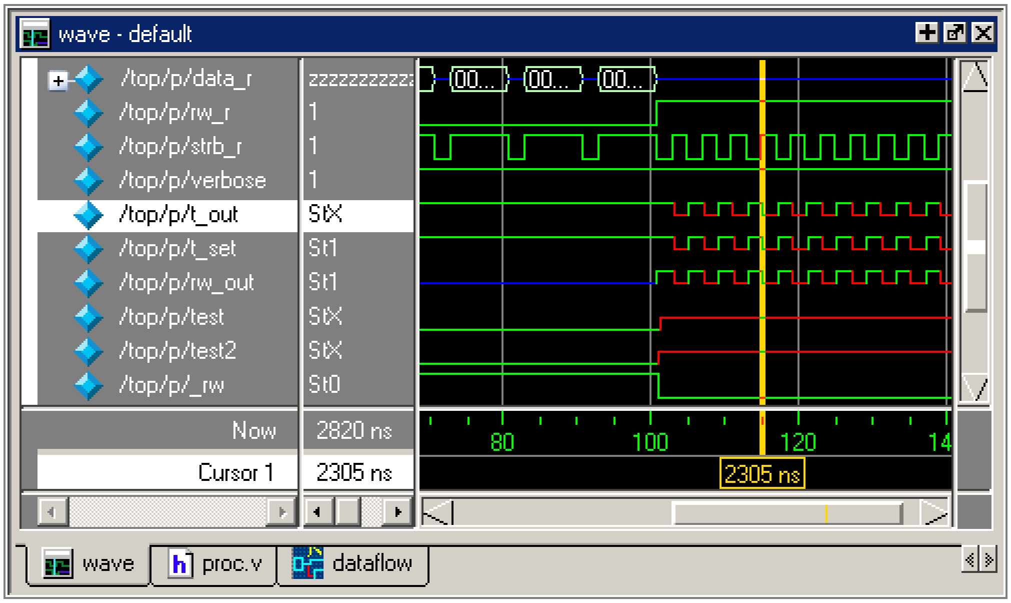
   The Dataflow display "jumps" to the source of the selected input event(s). The operation follows all signals selected in the wave viewer pane. You can change which signals are followed by changing the selection.
8. To continue tracing, go back to step 5 and repeat.

If you want to start over at the originally selected output, select **Trace** > **Trace event reset**.

Tracing the Source of an Unknown State (StX)

Another useful Dataflow window debugging tool is the ability to trace an unknown state (StX) back to its source. Unknown values are indicated by red lines in the Wave window (Figure 13-4) and in the wave viewer of the Dataflow window.

Figure 13-4. Unknown states shown as red lines in Wave window



The procedure for tracing to the source of an unknown state in the Dataflow window is as follows:

1. Load your design.

2. Log all signals in the design or any signals that may possibly contribute to the unknown value (**log -r /*** will log all signals in the design).

3. Add signals to the Wave window or wave viewer pane, and run your design the desired length of time.

4. Put a Wave window cursor on the time at which the signal value is unknown (StX). In Figure 13-4, Cursor 1 at time 2305 shows an unknown state on signal *t_out*.

5. Add the signal of interest to the Dataflow window by doing one of the following:

   - double-clicking on the signal's waveform in the Wave window,

   - right-clicking the signal in the Objects window and selecting **Add to Dataflow > Selected Signals** from the popup menu,

   - selecting the signal in the Objects window and selecting **Add > Dataflow > Selected Signals** from the menu bar.

6. In the Dataflow window, make sure the signal of interest is selected.

7. Trace to the source of the unknown by doing one of the following:

   - If the Dataflow window is docked, select **Tools > Trace > TraceX**, **Tools > Trace > TraceX Delay, Tools > Trace > ChaseX**, or **Tools > Trace > ChaseX Delay**.

   - If the Dataflow window is undocked, select **Trace > TraceX**, **Trace > TraceX Delay, Trace > ChaseX**, or **Trace > ChaseX Delay**.

     These commands behave as follows:

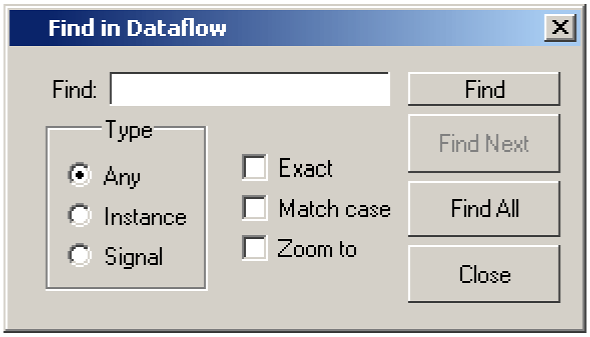
     - **TraceX / TraceX Delay**— Steps back to the last driver of an X value. **TraceX Delay** works similarly but it steps back in time to the last driver of an X value. **TraceX** should be used for RTL designs; **TraceX Delay** should be used for gate-level netlists with back annotated delays.

     - **ChaseX / ChaseX Delay** — "Jumps" through a design from output to input, following X values. **ChaseX Delay** acts the same as **ChaseX** but also moves backwards in time to the point where the output value transitions to X. **ChaseX** should be used for RTL designs; **ChaseX Delay** should be used for gate-level netlists with back annotated delays.

Finding Objects by Name in the Dataflow Window

Select **Edit > Find** from the menu bar, or click the Find icon in the toolbar, to search for signal, net, or register names or an instance of a component. This opens the Find in Dataflow dialog (Figure 13-5).

Figure 13-5. Find in Dataflow dialog



Type

● Any
○ Instance
○ Signal

□ Exact
□ Match case
□ Zoom to

With the Find in Dataflow dialog you can limit the search by type to instances or signals. You select Exact to find an item that exactly matches the entry you've typed in the Find field. The Match case selection will enforce case-sensitive matching of your entry. And the Zoom to selection will zoom in to the item in Find field.

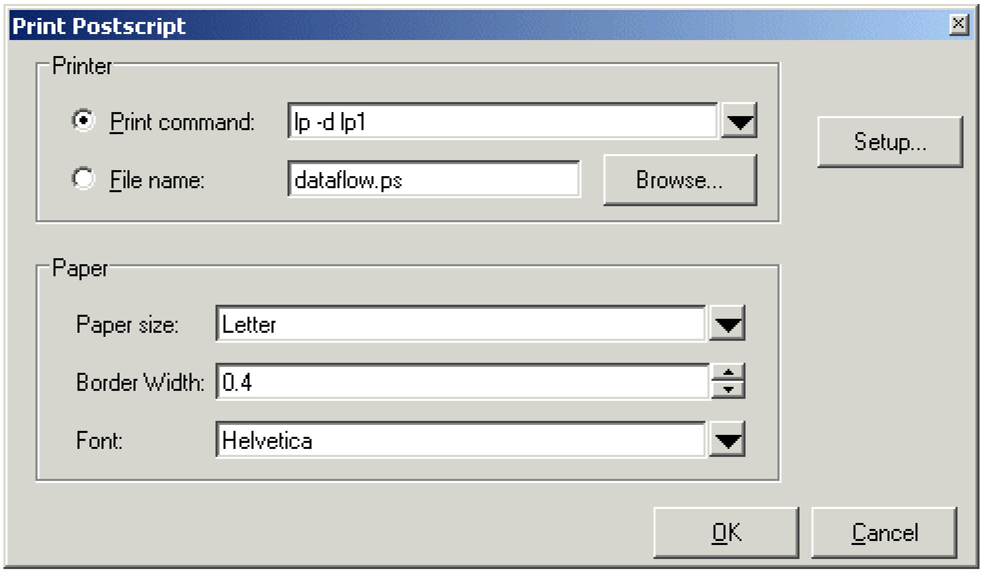
The Find All button allows you to find and highlight all occurrences of the item in the Find field. If Zoom to is checked, the view will change such that all selected items are viewable. If Zoom to is not selected, then no change is made to zoom or scroll state.

Printing and Saving the Display

Saving a .eps File and Printing the Dataflow Display from UNIX

Select **File** > **Print Postscript** to setup and print the Dataflow display in UNIX, or save the waveform as a .eps file on any platform.

Figure 13-6. The Print Postscript dialog



● Print command: lp -d lp1

○ File name: dataflow.ps

Paper size: Letter

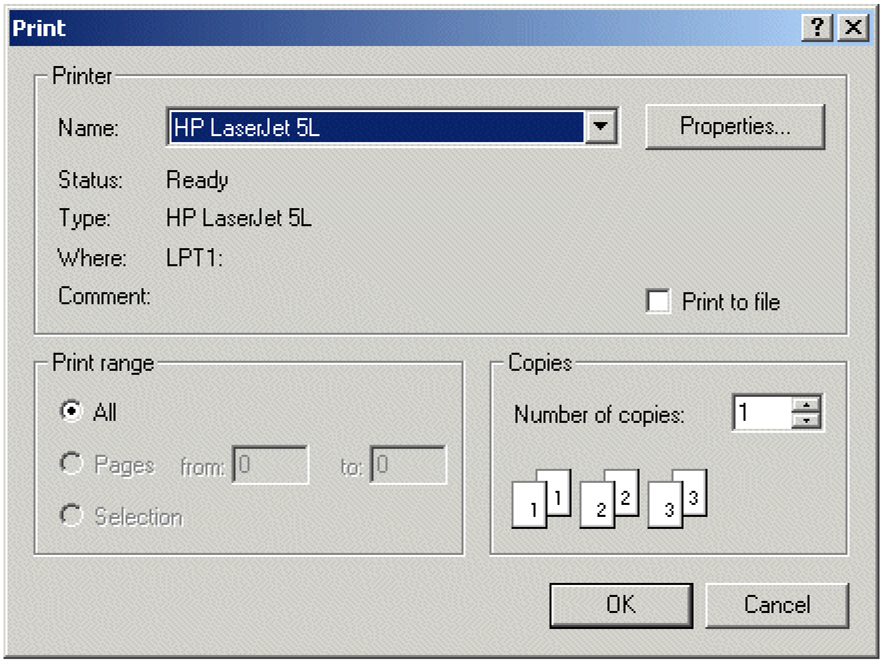
Border Width: 0.4

Font: Helvetica

Printing from the Dataflow Display on Windows Platforms

Select **File** > **Print** to print the Dataflow display or to save the display to a file.

Figure 13-7. The Print dialog



Name: HP LaserJet 5L

Status: Ready

Type: HP LaserJet 5L

Where: LPT1:

Comment:

□ Print to file

● All

○ Pages from: 0 to: 0

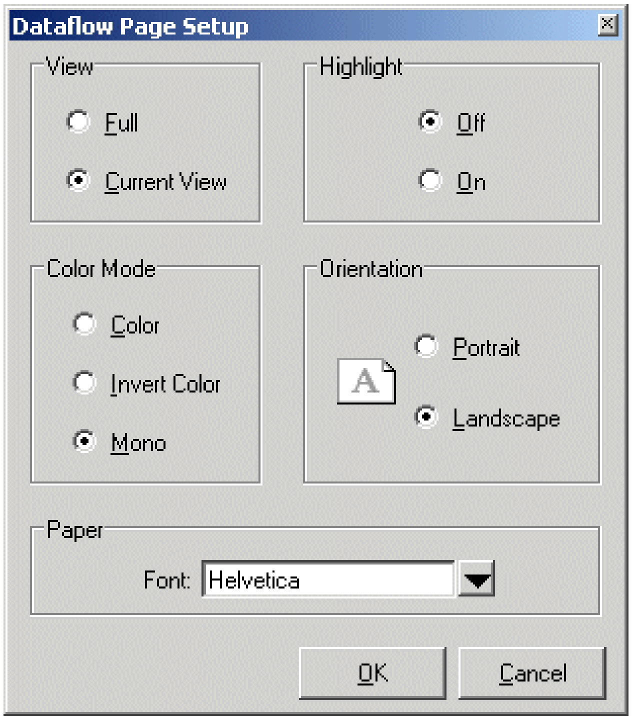
○ Selection

Number of copies: 1

Configuring Page Setup

Clicking the Setup button in the Print Postscript or Print dialog box allows you to configure page view, highlight, color mode, orientation, and paper options (this is the same dialog that opens via **File** > **Page setup**).

Figure 13-8. The Dataflow Page Setup dialog



View
○ Full
● Current View

Highlight
● Off
○ On

Color Mode
○ Color
○ Invert Color
● Mono

Orientation
○ Portrait
● Landscape

Paper
Font: Helvetica

Symbol Mapping

The Dataflow window has built-in mappings for all Verilog primitive gates (i.e., AND, OR, etc.). For components other than Verilog primitives, you can define a mapping between processes and built-in symbols. This is done through a file containing name pairs, one per line, where the first name is the concatenation of the design unit and process names, (DUname.Processname), and the second name is the name of a built-in symbol. For example:

```
xorg(only).p1 XOR
org(only).p1 OR
andg(only).p1 AND
```

Entities and modules are mapped the same way:

```
AND1 AND
AND2 AND  # A 2-input and gate
AND3 AND
AND4 AND
AND5 AND
AND6 AND
xnor(test) XNOR
```

Note that for primitive gate symbols, pin mapping is automatic.

The Dataflow window looks in the current working directory and inside each library referenced by the design for the file *dataflow.bsm* (.bsm stands for "Built-in Symbol Map"). It will read all files found.

User-defined symbols

You can also define your own symbols using an ASCII symbol library file format for defining symbol shapes. This capability is delivered via Concept Engineering's Nlview^{TM} widget Symlib format. For more specific details on this widget, see

www.model.com/support/documentation/BOOK/nlviewSymlib.pdf.

The Dataflow window will search the current working directory, and inside each library referenced by the design, for the file *dataflow.sym*. Any and all files found will be given to the Nlview widget to use for symbol lookups. Again, as with the built-in symbols, the DU name and optional process name is used for the symbol lookup. Here's an example of a symbol for a full adder:

```
symbol adder(structural) * DEF \
   port a in -loc -12 -15 0 -15 \
   pinattrdsp @name -cl 2 -15 8 \
   port b in -loc -12 15 0 15 \
   pinattrdsp @name -cl 2 15 8 \
   port cin in -loc 20 -40 20 -28 \
   pinattrdsp @name -uc 19 -26 8 \
   port cout out -loc 20 40 20 28 \
pinattrdsp @name -lc 19 26 8 \
   port sum out -loc 63 0 51 0 \
   pinattrdsp @name -cr 49 0 8 \
   path 10 0 0 7 \
   path 0 7 0 35 \
   path 0 35 51 17 \
   path 51 17 51 -17 \
   path 51 -17 0 -35 \
   path 0 -35 0 -7 \
   path 0 -7 10 0
```

Port mapping is done by name for these symbols, so the port names in the symbol definition must match the port names of the Entity|Module|Process (in the case of the process, it's the signal names that the process reads/writes).

Note

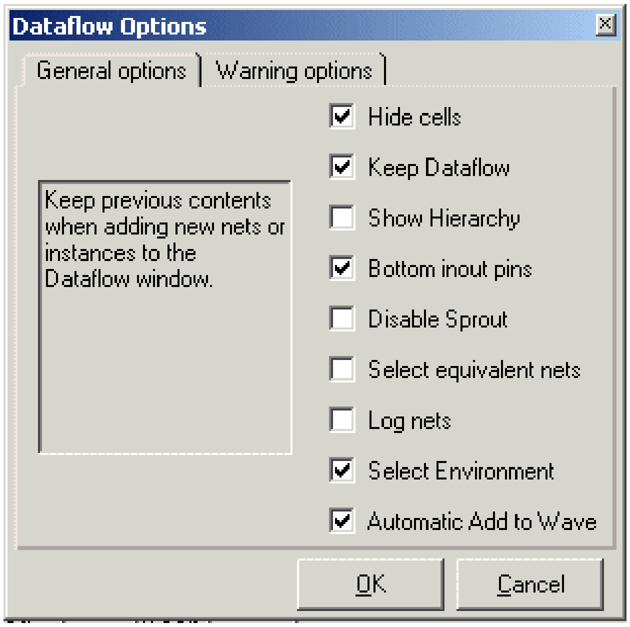
When you create or modify a symlib file, you must generate a file index. This index is how the Nlview widget finds and extracts symbols from the file. To generate the index, select **Tools** > **Create symlib index** (Dataflow window) and specify the symlib file. The file will be rewritten with a correct, up-to-date index.

Configuring Window Options

You can configure several options that determine how the Dataflow window behaves. The settings affect only the current session.

Select **Tools** > **Options** to open the Dataflow Options dialog box.

Figure 13-9. Configuring Dataflow options



Keep previous contents when adding new nets or instances to the Dataflow window.

☑ Hide cells
☑ Keep Dataflow
☐ Show Hierarchy
☑ Bottom inout pins
☐ Disable Sprout
☐ Select equivalent nets
☐ Log nets
☑ Select Environment
☑ Automatic Add to Wave

Chapter 14
Measuring Coverage

Note

The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Most commands related to coverage are used in one of three modes that correspond to the type of coverage analysis required.

Table 14-1.

Mode	Type of Coverage Analysis	Commands
Simulation Mode	Interactive simulation	**coverage** and **vcover** commands (such as **clear**, **merge**, **report**, **save**...)
Coverage View Mode	Post-process	**vsim -viewcov <file>.ucdb** brings up separate GUI All coverage commands are available
Batch Mode	Batch simulation	**vcover** utility with commands given as arguments to the invocation

Each of these modes of analysis act upon a single, universal database that stores your coverage data, the Unified Coverage Database.

Unified Coverage Database

The Unified Coverage DataBase (UCDB) is a custom database for representing coverage data. Coverage data collected in the database includes:

- Code coverage: branch, condition, expression, statement, and toggle
- Finite State Machine (FSM) coverage
- 0-In structural coverage
- SystemVerilog covergroup coverage
- SVA and PSL directive coverage

- Assertion data (pass, fail, attempt counts)
- User-defined data
- Test item data

0-In data is written into the database with a standalone converter. The UCDB is used natively by ModelSim for all coverage data, deprecating previous separate file formats for code coverage and functional coverage.

Purpose of the UCDB

The Unified Coverage Database is a single persistent form for various kinds of verification data, notably: coverage data of all kinds, and some other information useful for analyzing verification results.

When created from ModelSim, it is a single "snapshot" of data in the kernel. Thus, it represents all coverage and assertion objects in the design and testbench, along with enough hierarchical environment to indicate where these objects reside. This data is sufficient to generate complete coverage reports and can also be combined with data acquired outside ModelSim – e.g., 0-In coverage and user-defined data.

Important Notes About Coverage Statistics

You should be aware of the following special circumstances related to collecting coverage statistics:

- When ModelSim optimizes a design, it "removes" unnecessary lines of code (e.g., code in a procedure that is never called). The lines that are optimized away are not counted in the coverage data, and this may cause misleading results. When you compile with coverage enabled, ModelSim disables certain optimizations depending on which coverage types you choose. This produces more accurate statistics but also may slow simulation.

  The table below shows the coverage types and the effect on optimizations when compiling.

Table 14-2.

Coverage type	Effect on optimizations
statement	optimizations not disabled automatically; specify -O0 to get most accurate statistics
branch	case statement optimizations are disabled automatically
condition	optimizations not disabled automatically
expression	all optimizations disabled automatically

Table 14-2.

Coverage type	Effect on optimizations
toggle	optimizations not disabled automatically
fsm	optimizations not disabled automatically

Optimizations can be disabled during simulation with vopt -coverage or vsim -coverage. When either of these command options are used the following occurs:

- lines are visible
- verilog nets and registers are visible
- inlining is turned off
- process merging is turned off
- state machine variables are not affected

The VoptCoverageOptions *modelsim.ini* variable also disables optimizations that interfere with the collection of coverage statistics.

- Package bodies are not instance-specific: ModelSim sums the counts for all invocations no matter who the caller is. Also, all standard and accelerated packages are ignored for coverage statistics calculation.
- Design units compiled with **-nodebug** are ignored, as if they were excluded.
- When condition coverage is enabled, expression short circuiting in VHDL is disabled. This can result in simulation errors. For example:

  ```
  if ( (i /= 0) AND (10/i > 1) ) then
  ```

  will never produce a divide-by-0 error in normal VHDL because the test *(i/=0)* will be FALSE if *i* is 0 and *10/i* will never be done. With condition coverage enabled both expressions *(i /= 0)* and *(10/i > 1)* will always be evaluated and if *i* is 0, there will be a divide-by-zero error reported.
- Coverage data is not collected for generate blocks.
- You may find that design units or instances excluded from code coverage will appear in toggle coverage statistics reports. This happens when ports of the design unit or instance are connected to nets that have toggle coverage turned on elsewhere in the design.

Coverage View Mode

Raw UCDB coverage data can be saved, merged with coverage statistics from the current simulation or from previously saved coverage data, and viewed at a later date using the coverage view mode. You can save and merge coverage statistics and invoke the tool in standalone, coverage view mode via the command line or the **$coverage_save** Verilog system task (see System Tasks and Functions Specific to the Tool).

The coverage view mode allows all ModelSim coverage data saved in the UCDB format – code coverage, covergroup coverage, directive coverage, and assertion data – to be called up and displayed in the same coverage GUIs used for simulation. The coverage view invocation of the tool is separate from that of the simulation.

Saving Coverage Data

To use the coverage view mode, coverage data must have been previously saved in the UCDB format with the coverage save command or with the $coverage_save Verilog system task (see System Tasks and Functions Specific to the Tool). For example:

```
coverage save myfile1.ucdb
```

saves coverage data from the current simulation into a UCDB file called *myfile1.ucdb*.

The **coverage save** command preserves instance-specific information.

Merging Coverage Data

The merge utility, vcover merge, allows you to merge sets of coverage data without first loading a design. The merge utility is a standard ModelSim utility that can be invoked from the command line. For example,

```
vcover merge myresult myfile1 myfile2
```

merges coverage statistics in UCDB files *myfile1* and *myfile2* and writes them to a new UCDB file called *myresult*.

If an instance in a code coverage database has been changed, a warning will be generated. Warnings can be disabled with the -quiet option.

The **vcover** code coverage utility supports instance-specific toggles and summing the instances of each design unit. It also supports source code annotation and printing of condition and expression truth tables.

Invoking the Coverage View Mode

Use the vsim command with the -viewcov <ucdb_filename> option to invoke the coverage view mode. This allows you to view saved and/or merged coverage results from earlier simulations.

The coverage view mode is invoked with:

```
vsim -viewcov myresult.ucdb
```

where *myresult.ucdb* is the coverage data saved in ucdb format. The design hierarchy and coverage data is imported from a UCDB.

Coverage view mode may be combined with WLF view mode, in which case some coverage objects (directives and assertions) may be debugged through both browser GUIs and the wave window.

Merging Coverage Data

When capturing data over time, data must be merged, which does not perfectly preserve all data. (For example, a merged UCDB does not preserve exactly which tests incremented exactly which set of coverage items.) A merged UCDB is still a snapshot: a summary of the coverage that would exist if all merged tests were somehow run in the kernel, accumulating coverage by incrementing coverage counts, and the final values written to a single UCDB.

Thus, if you need to preserve all data over a span of time, the approach must be the simplest one: preserve multiple UCDBs over time, each one representing a faithful snapshot of a single test run. ModelSim does not provide facilities for managing multiple UCDBs, as operating system file systems can suffice.

Similarly, if snapshots are taken of slightly different hierarchy (because the testbench is different, or the design resides in a different locale in testbench hierarchy) this must be taken into account in the merge: the hierarchies must somehow be rationalized during the merge operation.

Merging coverage data in previous versions of ModelSim was assumed to be by instance path. In other words, you had to know that in the case of two hierarchies, */top/design* and */top/i/design*, it is the "design" instance that is of interest for merging. To deal with this, you had to strip off hierarchy with the **coverage merge -strip** option during merge and install new hierarchy with **coverage merge -install**. This allowed you to convert one hierarchy into another.

But this does not work when a design is instantiated twice in the same hierarchy. ModelSim cannot merge both instantiations into the same database. This is because, say, /top/designinst1 and /top/other/designinst2 are always distinct if they exist in the same coverage file. Because they are different paths they are always considered to be distinct.

A better approach would be to merge by design unit type. See the coverage merge command.

With this approach, a given design unit type (or set of types) completely specifies what subtrees of the UCDB to merge. Even in cases where each UCDB has one and only one instance of the given design unit type, this offers automatic determination of the -install and -strip arguments that could be used to perform the merge with the current system.

The resulting merge by design unit type yields a hierarchy with a single top-level module of the given type.

```
    vcover merge -bydu sometype file*.ucdb
```

Code Coverage

ModelSim code coverage gives you graphical and report file feedback on which statements, branches, conditions, and expressions in your source code have been executed. It also measures bits of logic that have been toggled during execution.

With coverage enabled, ModelSim counts how many times each executable statement, branch, condition, expression, and logic node in each instance is executed during simulation.

- **Statement** coverage counts the execution of each statement on a line individually, even if there are multiple statements in a line.
- **Branch** coverage counts the execution of each conditional "if/then/else" and "case" statement and indicates when a true or false condition has not executed.
- **Condition** coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage.
- **Expression** coverage analyzes the expressions on the right hand side of assignment statements, and is similar to condition coverage.
- **Toggle** coverage counts each time a logic node transitions from one state to another.

Coverage statistics are displayed in the Main, Objects, and Source windows and also can be output in different text reports (see Reporting Coverage Data). Raw coverage data can be saved and recalled, or merged with coverage data from the current simulation (see Coverage Statistics Details).

ModelSim code coverage offers these benefits:

- It is totally non-intrusive because it’s integrated into the ModelSim engine – it doesn’t require instrumented HDL code as do third-party coverage products.
- It has very little impact on simulation performance (typically 10 to 20 percent).
- It allows you to merge sets of coverage data without requiring elaboration of the design or a simulation license.

Usage Flow for Code Coverage

The following is an overview of the usage flow for simulating with code coverage. More detailed instructions are presented in the sections that follow.

1. Compile the design using the **-cover bcestxf** or **-coverAll** arguments to vcom or vlog. (See Enabling Code Coverage.)

2. Simulate the design using the **-coverage** argument to vsim.
3. Run the design.
4. Analyze coverage statistics in the Main, Objects, and Source windows.
5. Edit the source code to improve coverage.
6. Re-compile, re-simulate, and re-analyze the statistics and design.

Supported Types

ModelSim code coverage supports the following data types.

VHDL Coverage Support

Supported types for toggle coverage are boolean, bit, std_logic, enum, integer and arrays of these types. Counts are recorded for each enumeration value and a signal is considered “toggled” if all the enumerations have non-zero counts. For VHDL integers, a record is kept of each value the integer assumes and an associated count. The maximum number of values recorded is determined by a limit variable that can be changed on a per-signal basis. The default is 100 values. The limit variable can be turned off completely with the **-toggleNoIntegers** option for the vsim command. The limit variable can be increased by setting the **vsim** command line option **-toggleMaxIntValues**, setting ToggleMaxIntValues in the modelsim.ini file, or setting the Tcl variable **ToggleMaxIntValues**.

Condition and expression coverage supports bit and boolean types. Arbitrary types are supported when they involve a relational operator with a boolean result. These types of subexpressions are treated as an external expression that is first evaluated and then used as a boolean input to the full condition. The subexpression can look like:

```
(var <relop> const)
```

or:

```
(var1 <relop> var2)
```

where var, var1 and var2 may be of any type; <relop> is a relational operator (e.g.,==,<,>,>=); and const is a constant of the appropriate type.

Logical operators (e.g.,and,or,xor) are supported for std_logic/std_ulogic, bit, and boolean variable types.

Verilog/SystemVerilog Coverage Support

Supported types for toggle coverage are net, register, bit, enum and integer (which includes shortint, int, longint, byte, integer and time). SystemVerilog integer types are treated as 32-bit registers and counts are kept for each bit.

For condition and expression coverage, as in VHDL, arbitrary types are supported when they involve a relational operator with a boolean result.

Logical operator (e.g.,&&,||,^) are supported for one-bit net, logic, and reg types.

SystemC Coverage Support

Code coverage does not work on SystemC design units.

Enabling Code Coverage

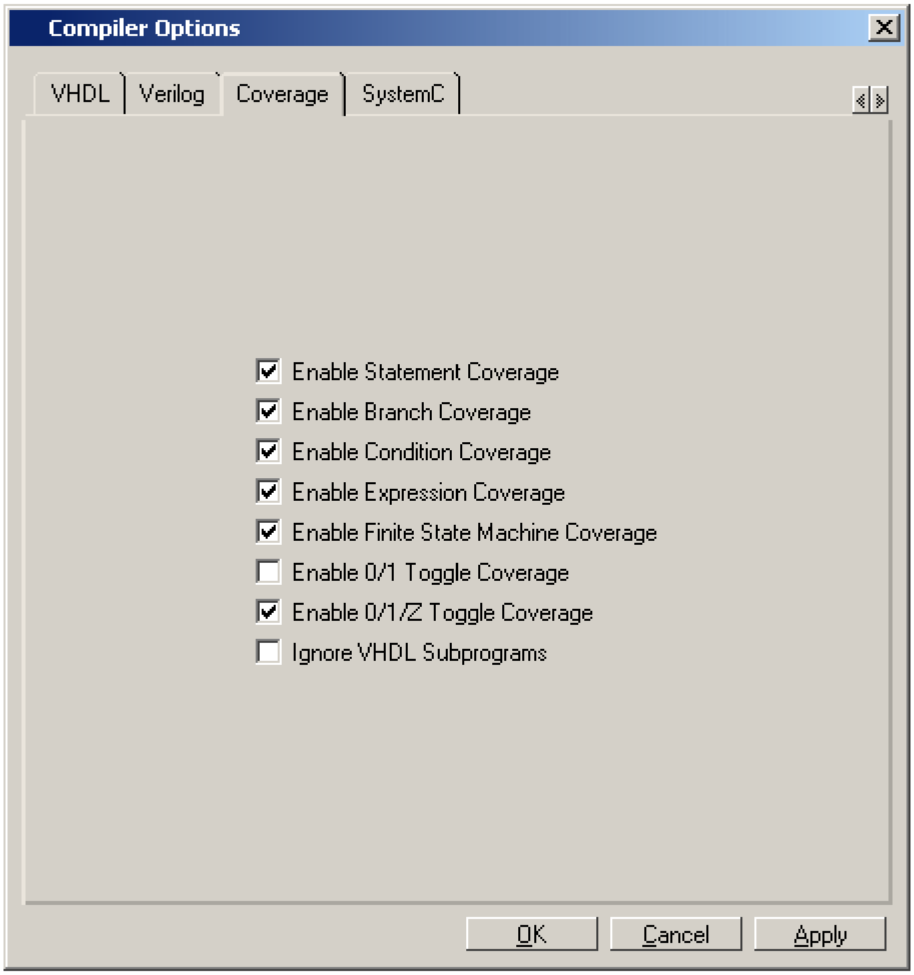
Enabling code coverage is a two-step process:

1. Use the **-cover** argument to **vcom** or **vlog** when you compile your design. This argument tells ModelSim which coverage statistics to collect. For example:

   ```
   vlog top.v proc.v cache.v -cover bcesxf
   ```

   Each character after the **-cover** argument identifies a type of coverage statistic: "**b**" indicates branch, "**c**" indicates condition, "**e**" indicates expression, "**s**" indicates statement, "**t**" indicates 2-transition toggle, "**x**" indicates extended 6-transition toggle coverage (t and x are mutually exclusive), and “**f**” indicates Finite State Machine coverage. See Enabling Toggle Coverage for details on two other methods for enabling toggle coverage.

You can use graphic interface to perform the same task. Select **Compile > Compile Options** and select the Coverage tab. Alternatively, if you are using a project, right-click on a selected design object (or objects) and select **Properties**.



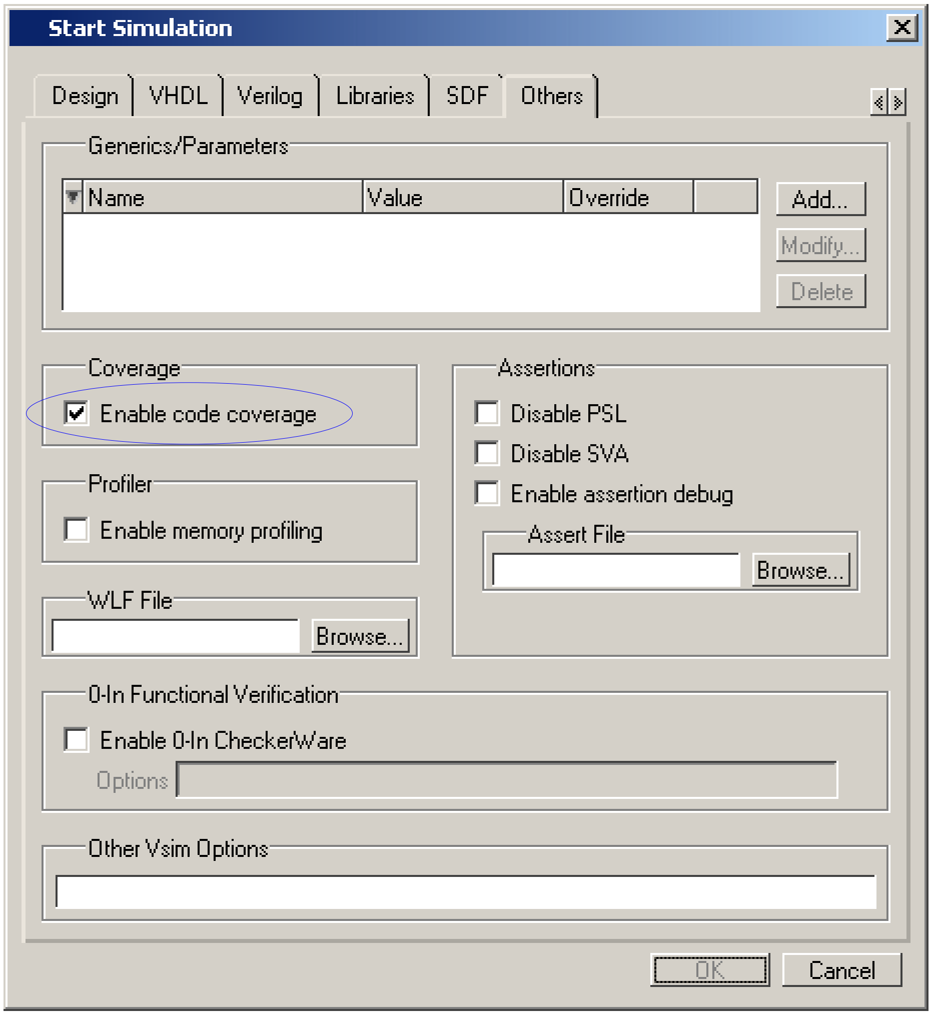
☑ Enable Statement Coverage
☑ Enable Branch Coverage
☑ Enable Condition Coverage
☑ Enable Expression Coverage
☑ Enable Finite State Machine Coverage
□ Enable 0/1 Toggle Coverage
☑ Enable 0/1/Z Toggle Coverage
□ Ignore VHDL Subprograms

If you check **Ignore VHDL Subprograms**, code coverage collection is disabled for VHDL subprograms.

2. Use the **-coverage** argument to **vsim** when you simulate your design. For example:

```
vsim -coverage work.top
```

Or, use the graphic interface. Select **Simulate > Start Simulation** and select the design unit to be simulated in the Design tab. Then select the Others tab and check **Enable code coverage** box as shown below.



☑ Enable code coverage

□ Enable memory profiling

□ Disable PSL

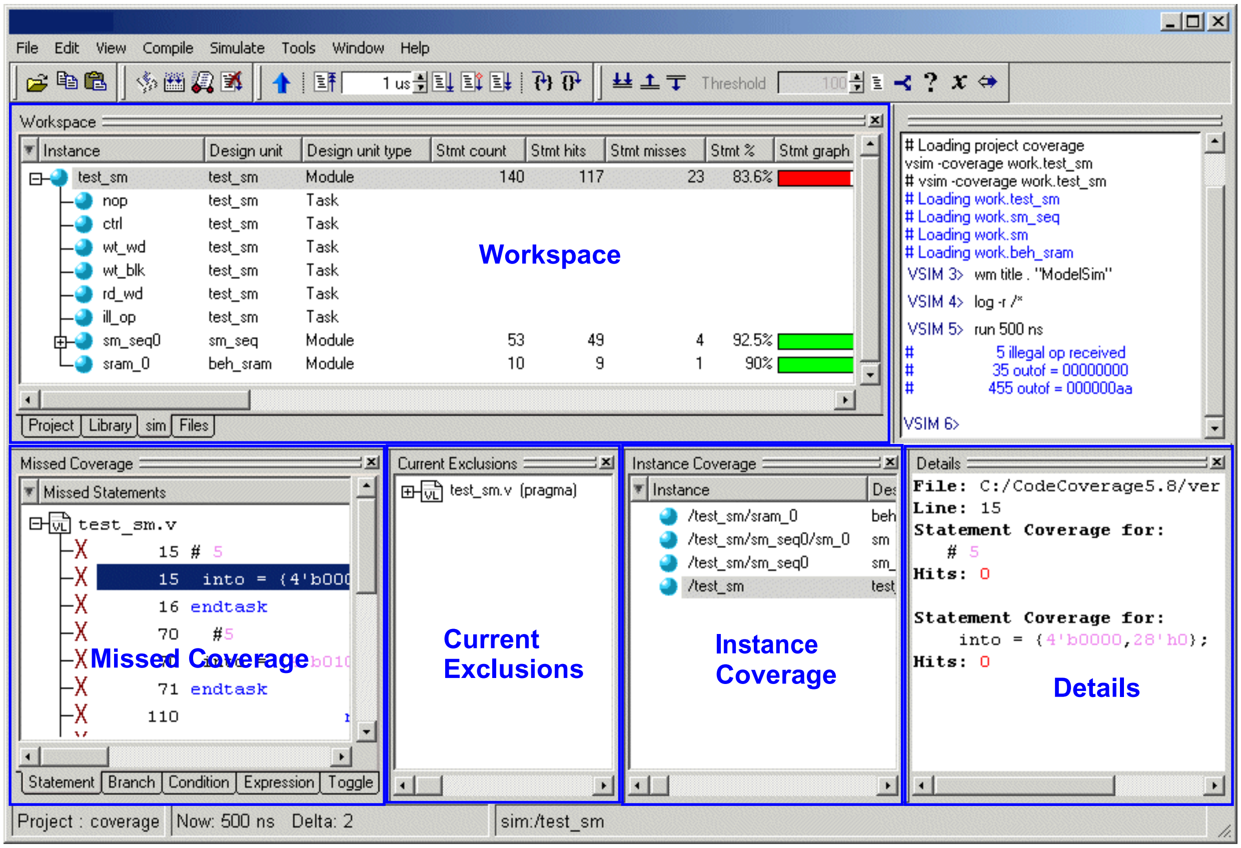
□ Disable SVA

□ Enable assertion debug

□ Enable 0-In CheckerWare

Viewing Coverage Data in the Main Window

When you simulate a design with code coverage enabled, coverage data is displayed in the Main, Source, and Objects windows. In the Main window, coverage data displays in five window panes: Workspace, Missed Coverage, Current Exclusions, Instance Coverage, and Details.



Instance	Design unit	Design unit type	Stmt count	Stmt hits	Stmt misses	Stmt %	Stmt graph
test_sm	test_sm	Module	140	117	23	83.6%	🟥
nop	test_sm	Task					
ctrl	test_sm	Task					
wt_wd	test_sm	Task					
wt_blk	test_sm	Task					
rd_wd	test_sm	Task					
ill_op	test_sm	Task					
sm_seq0	sm_seq	Module	53	49	4	92.5%	🟩
sram_0	beh_sram	Module	10	9	1	90%	🟩

```
# Loading project coverage
vsim -coverage work.test_sm
# vsim -coverage work.test_sm
# Loading work.test_sm
# Loading work.sm_seq
# Loading work.sm
# Loading work.beh_sram
VSIM 3> wm title . "ModelSim"
VSIM 4> log -r /*
VSIM 5> run 500 ns
#              5 illegal op received
#              35 outof = 00000000
#              455 outof = 000000aa
VSIM 6>
```

```
test_sm.v
X      15  # 5
X      15   into = {4'b000
X      16  endtask
X      70    #5
X      70   into = {4'b010
X      71  endtask
X     110
```

```
test_sm.v  (pragma)
```

Instance	Des
/test_sm/sram_0	beh
/test_sm/sm_seq0/sm_0	sm
/test_sm/sm_seq0	sm_
/test_sm	test

```
File:  C:/CodeCoverage5.8/ver
Line:  15
Statement Coverage for:
    # 5
Hits:  0

Statement Coverage for:
     into = {4'b0000,28'h0};
Hits:  0
```

Project : coverage   Now: 500 ns   Delta: 2   sim:/test_sm

The table below summarizes the Main window coverage panes. For further details, see Code Coverage Panes.

Table 14-3.

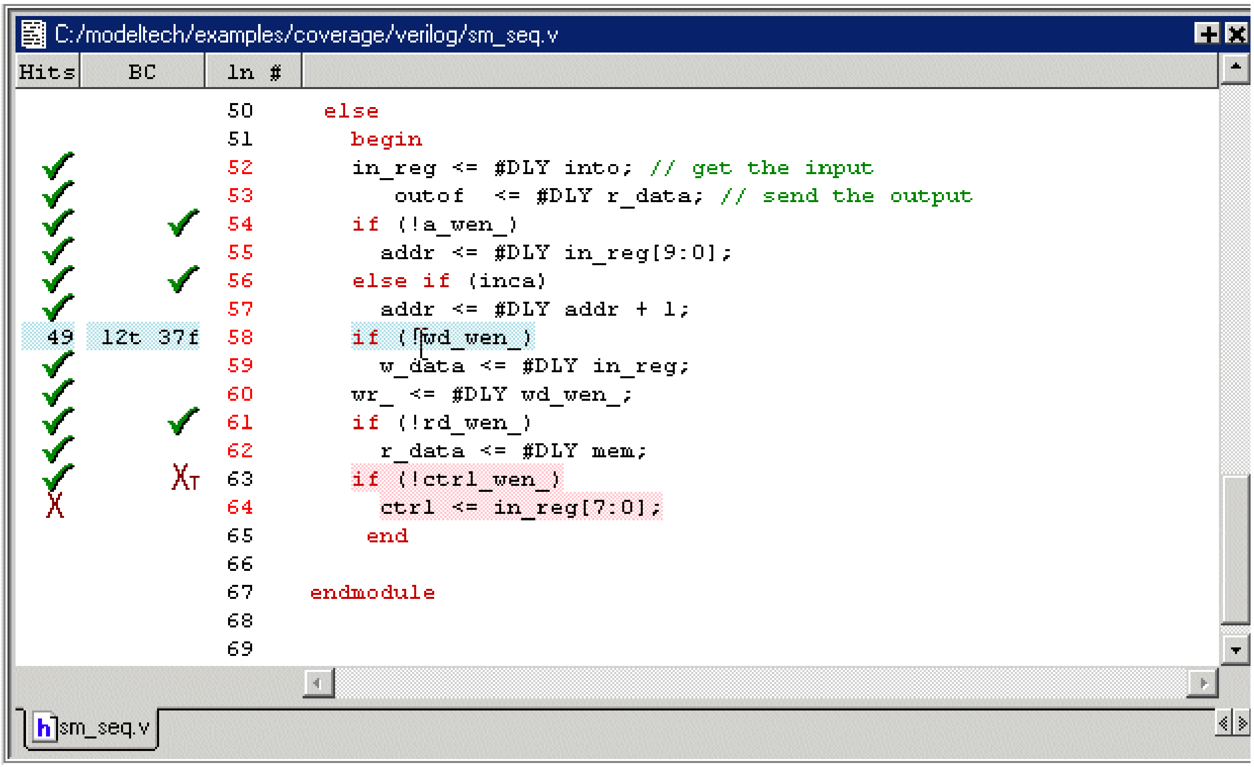
Coverage pane	Description
Workspace	displays coverage data and graphs for each design object or file
Missed Coverage	displays missed coverage for the selected design object or file
Current exclusions[1]	lists all files and lines that are excluded from the current analysis
Instance coverage	displays coverage statistics for each instance in a flat format
Details	displays details of missed coverage and toggle coverage

[1] The Current Exclusions pane does not display by default. Select **View** > **Code Coverage** > **Current Exclusions** to display the pane.

Viewing Coverage Data in the Source Window

The Source Window includes two columns for code coverage statistics – the Hits column and the BC (Branch Coverage) column. These columns provide an immediate visual indication about how your source code is executing. The default code coverage indicators are check marks and Xs.

- A green check mark indicates that the statements, branches or expressions in a particular line have been covered.
- A red X indicates that a statement or branch was not covered.
- An X_T indicates the true branch of an conditional statement was not covered.
- An X_F indicates the false branch was not covered.
- A green "E" indicates a line of code that has been excluded from code coverage statistics.



C:/modeltech/examples/coverage/verilog/sm_seq.v

```
Hits  BC        ln #
                50     else
                51       begin
 ✓              52       in_reg <= #DLY into; // get the input
 ✓              53          outof  <= #DLY r_data; // send the output
 ✓       ✓      54       if (!a_wen_)
 ✓              55         addr <= #DLY in_reg[9:0];
 ✓       ✓      56       else if (inca)
 ✓              57         addr <= #DLY addr + 1;
49    12t 37f   58       if (!wd_wen_)
 ✓              59         w_data <= #DLY in_reg;
 ✓              60       wr_  <= #DLY wd_wen_;
 ✓       ✓      61       if (!rd_wen_)
 ✓              62         r_data <= #DLY mem;
 ✓       XT     63       if (!ctrl_wen_)
 X              64         ctrl <= in_reg[7:0];
                65        end
                66
                67    endmodule
                68
                69
```

Expressions have associated truth tables that can be seen in the Details pane when an expression is selected in the Missed Coverage pane. Each line in the truth table is one of the possible combinations for the expression. The expression is considered to be covered (gets a green check mark) only if the entire truth table is covered.

When you hover the cursor over a line of code (see line 58 in the illustration above), the number of statement and branch executions, or "hits," will be displayed in place of the check marks and

Xs. If you prefer, you can display only numbers by selecting **Tools > Code Coverage > Show Coverage Numbers**.

Also, when you click in either the Hits or BC column, the Details pane in the Main window updates to display information on that line.

You can skip to "missed lines" three ways: select **Edit > Previous Coverage Miss** and **Edit > Next Coverage Miss** from the menu bar; click the Previous zero hits and Next zero hits icons on the toolbar; or press Shift-Tab (previous miss) or Tab (next miss).

Controlling Data Display in a Source Window

The **Tools** > **Code Coverage** menu contains several commands for controlling coverage data display in a Source window.

- **Hide/Show coverage data** toggles the *Hits* column off and on.
- **Hide/Show branch coverage** toggles the *BC* column off and on.
- **Hide/Show coverage numbers** displays the number of executions in the *Hits* and *BC* columns rather than checkmarks and Xs. When multiple statements occur on a single line an ellipsis ("...") replaces the Hits number. In such cases, hover the cursor over each statement to highlight it and display the number of executions for that statement.
- **Show coverage By Instance** displays only the number of executions for the currently selected instance in the Main window workspace.

Toggle Coverage

Toggle coverage is the ability to count and collect changes of state on specified nodes, including Verilog nets and registers and the following VHDL signal types: bit, bit_vector, std_logic, and std_logic_vector. Toggle coverage is integrated as a metric into the coverage tool so that the use model and reporting are the same as the other coverage metrics.

There are two modes of toggle coverage operation - standard and extended. Standard toggle coverage only counts Low or 0 <--> High or 1 transitions. Extended toggle coverage counts these transitions plus the following:

```
Z <--> 1 or H
Z <--> 0 or L
```

Extended coverage allows a more detailed view of testbench effectiveness and is especially useful for examining coverage of tri-state signals. It helps to ensure, for example, that a bus has toggled from high 'Z' to a '1' or '0', and a '1' or '0' back to a high 'Z'.

Toggle coverage ignores zero-delay glitches.

Enabling Toggle Coverage

In the section Enabling Code Coverage we explained that toggle coverage could be enabled during compile by using the ’t’ or ’x’ arguments with **vcom -cover** or **vlog -cover**. This section describes two other methods for enabling toggle coverage:

1. using the toggle add command
2. using the **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** selections in the Main window menu.

Using the Toggle Add Command

The toggle add command allows you to initiate toggle coverage at any time from the command line. Upon the next running of the simulation, toggle coverage data will be collected according to the arguments employed (i.e., the **-full** argument enables collection of extended toggle coverage statistics for the transitions mentioned above).

If you use a toggle add command on a group of signals to get standard toggle coverage, then try to convert to extended toggle coverage with the **toggle add -full** command on the same signals, nothing will change. The only way to change the internal toggle triggers from standard to extended toggle coverage is to restart vsim and use the correct command.

Using the Main Window Menu Selections

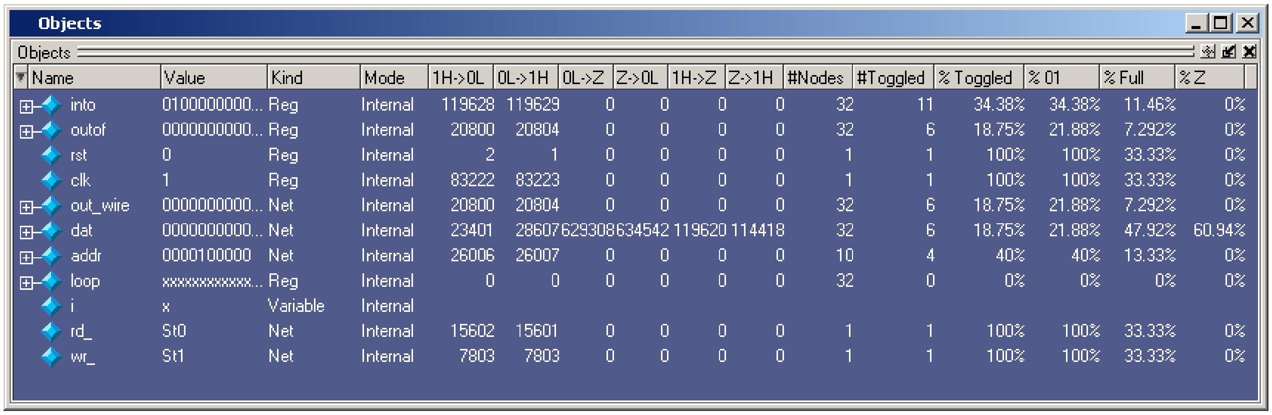
You can enable toggle coverage by selecting **Tools > Toggle Coverage > Add** or **Tools > Toggle Coverage > Extended** from the Main window menu. These selections allow you to enable toggle coverage for Selected Signals, Signals in Region, or Signals in Design.

After making a selection, toggle coverage statistics will be captured the next time you run the simulation.

Viewing Toggle Coverage Data in the Objects Pane

To view toggle coverage data in the Objects pane right-click in the pane to open a context popup menu the **Toggle Coverage** selection. When highlighted, this selection allows you to display toggle coverage data for the **Selected Signals**, the **Signals in Region**, or the **Signals in Design**.

Toggle coverage data is displayed in the Objects pane in multiple columns, as shown below. There is a column for each of the six transition types. Right click any column name to toggle that column on or off. See Objects Pane Toggle Coverage for more details on each column.



Name	Value	Kind	Mode	1H->0L	0L->1H	0L->Z	Z->0L	1H->Z	Z->1H	#Nodes	#Toggled	% Toggled	% 01	% Full	% Z
into	0100000000...	Reg	Internal	119628	119629	0	0	0	0	32	11	34.38%	34.38%	11.46%	0%
outof	0000000000...	Reg	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
rst	0	Reg	Internal	2	1	0	0	0	0	1	1	100%	100%	33.33%	0%
clk	1	Reg	Internal	83222	83223	0	0	0	0	1	1	100%	100%	33.33%	0%
out_wire	0000000000...	Net	Internal	20800	20804	0	0	0	0	32	6	18.75%	21.88%	7.292%	0%
dat	0000000000...	Net	Internal	23401	28607	629308	634542	119620	114418	32	6	18.75%	21.88%	47.92%	60.94%
addr	0000100000	Net	Internal	26006	26007	0	0	0	0	10	4	40%	40%	13.33%	0%
loop	xxxxxxxxxxxx...	Reg	Internal	0	0	0	0	0	0	32	0	0%	0%	0%	0%
i	x	Variable	Internal												
rd_	St0	Net	Internal	15602	15601	0	0	0	0	1	1	100%	100%	33.33%	0%
wr_	St1	Net	Internal	7803	7803	0	0	0	0	1	1	100%	100%	33.33%	0%

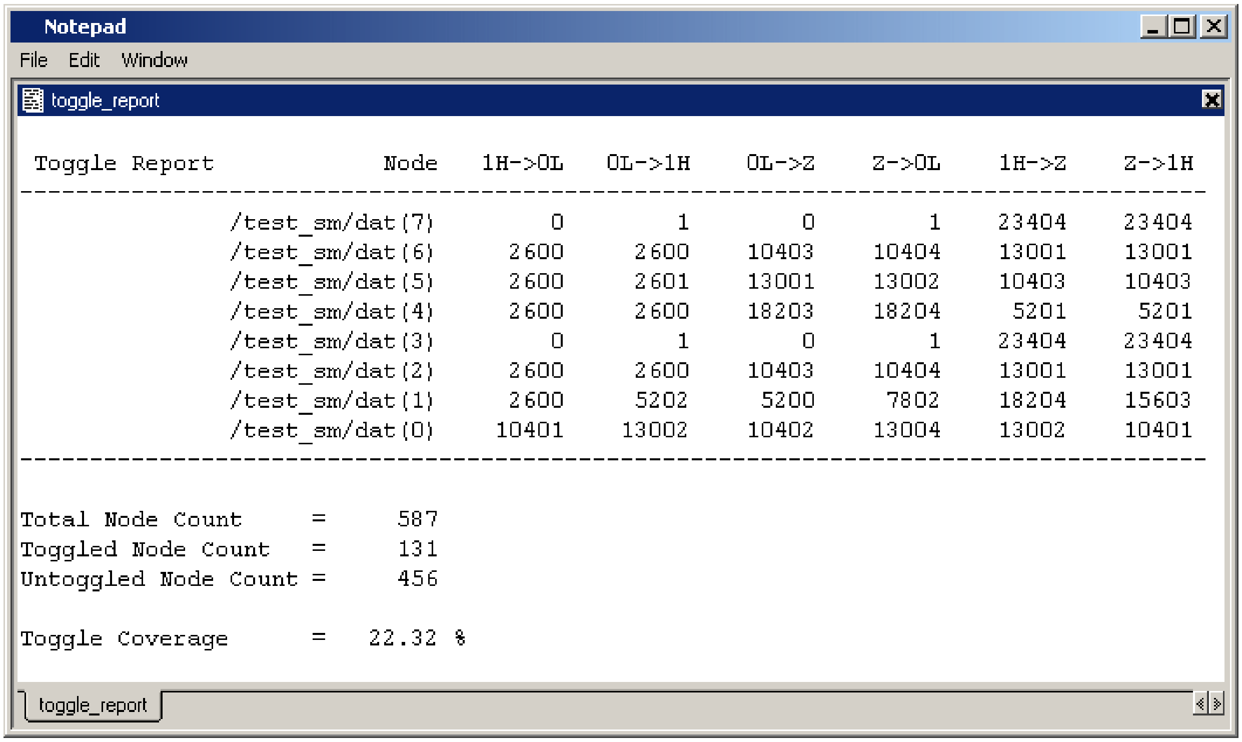
Toggle Coverage Reporting

The toggle report command displays a list of all nodes that have not transitioned to both 0 and 1 at least once, and the counts for how many times each node toggled for each state transition type. Also displayed is a summary of the number of nodes checked, the number that toggled, the number that didn't toggle, and a percentage that toggled.

The **toggle report** command is intended to be used as follows:

1. Enable statistics collection with the toggle add command.
2. Run the simulation with the run command.

3. Produce the report with the **toggle report** command..



```
 Toggle Report                Node   1H->0L   0L->1H    0L->Z    Z->0L    1H->Z    Z->1H
-------------------------------------------------------------------------------------
              /test_sm/dat(7)        0        1        0        1    23404    23404
              /test_sm/dat(6)     2600     2600    10403    10404    13001    13001
              /test_sm/dat(5)     2600     2601    13001    13002    10403    10403
              /test_sm/dat(4)     2600     2600    18203    18204     5201     5201
              /test_sm/dat(3)        0        1        0        1    23404    23404
              /test_sm/dat(2)     2600     2600    10403    10404    13001    13001
              /test_sm/dat(1)     2600     5202     5200     7802    18204    15603
              /test_sm/dat(0)    10401    13002    10402    13004    13002    10401
-------------------------------------------------------------------------------------

Total Node Count     =     587
Toggled Node Count   =     131
Untoggled Node Count =     456

Toggle Coverage      =   22.32 %
```

You can produce this same information using the coverage report command.

Port Collapsing and Toggle Coverage

The simulator collapses certain ports that are connected to the same signal in order to improve performance, and those collapsed signals will not appear in the report. If you want to ensure that you are reporting all signals in the design, use the **-nocollapse** argument to **vsim** when you load your design. The **-nocollapse** argument degrades simulator performance, so it should be used only when it is absolutely necessary to see all signals in a toggle report.

Excluding Nodes from Toggle Coverage

You can disable toggle coverage with the toggle disable command. This command disables toggle statistics collection on the specified nodes and provides a method of implementing coverage exclusions for toggle coverage. It is intended to be used as follows:

1. Enable toggle statistics collection for all signals using the **-cover t/x** argument to vcom or vlog.
2. Exclude certain signals by disabling them with the toggle disable command.

The toggle enable command re-enables toggle statistics collection on nodes whose toggle coverage has previously been disabled via the **toggle disable** command. (See the Reference Manual for correct syntax.)

Excluding Bus Bits from Toggle Coverage

You can exclude bus bits from toggle coverage using special source code pragmas.

- Verilog syntax

  ```
  // coverage toggle_ignore <simple_signal_name> "{<list> | all}"
  ```

- VHDL syntax

  ```
  -- coverage toggle_ignore <simple_signal_name> "{<list> | all}"
  ```

The following rules apply to using these pragmas:

- <list> is a space-separated list of bit indices or ranges, where a range is two integers separated by ':' or '-'.
- If using a range, the range must be in the same ascending or descending order as the signal declaration.
- The “all” keyword indicates that you are excluding the entire signal and can be specified instead of “<list>”.
- Quotes are required around the <list> or all keyword.
- The pragma must be placed within the declarative region of the module or architecture in which <simple_signal_name> is declared.

Excluding enum Signals from Toggle Statistics

You can exclude individual VHDL enums or ranges of enums from toggle coverage and reporting by specifying enum exclusions in source code pragmas or by using the **-exclude** argument to the toggle add command.

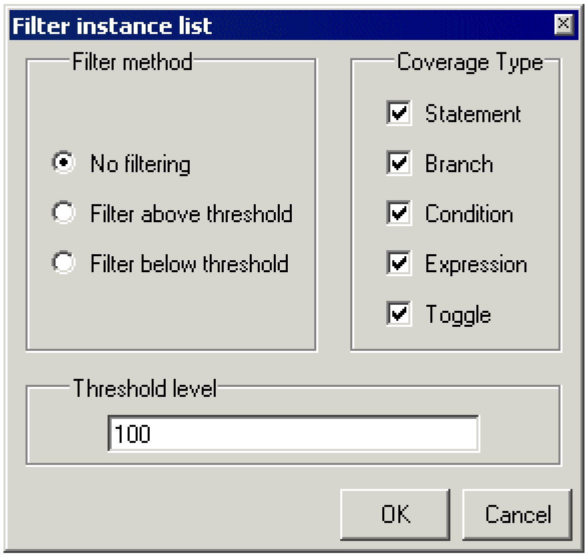
Finite State Machine Coverage

Information about Finite State Machine coverage can be found in the Finite State Machines chapter.

Setting a Coverage Threshold

You can specify a percentage above or below which you don’t want to see coverage statistics. For example, you might set a threshold of 85% such that only objects with coverage below that percentage are displayed. Anything above that percentage is filtered.

You can set a filter using either a dialog or toolbar icons (see below). To access the dialog, right-click any object in the Instance Coverage pane and select **Set filter**.



Filter instance list

Filter method

● No filtering
○ Filter above threshold
○ Filter below threshold

Coverage Type

☑ Statement
☑ Branch
☑ Condition
☑ Expression
☑ Toggle

Threshold level: 100

Managing Exclusions

ModelSim includes the following mechanisms for creating coverage exclusions:

- You can use fcover configure -exclude/-include to exclude functional coverage SVA and PSL directives.
- Use source code pragmas to exclude individual code coverage metrics.
- Use the coverage exclude command for code coverage exlcusions.
- Use the GUI to create exclusions. Right-click any object in the Missed Coverage pane (except Toggle and FSM objects) and select Exclude Selection

Exclusions are stored in the current exclusions file and in the UCDB. This allows report generation based on the most recent snapshot of functional coverage state. In its internal operation, ModelSim guarantees that the exclusions file and the UCDB are consistent with each other in the exclusions specified.

Exclusions are implemented using the "flags" field associated with a cover item. The primary functions *ucdb_IncludeCover* and *ucdb_ExcludeCover* dynamically include or exclude cover items. The flag UCDB_EXCLUDE_PRAGMA is used as a flag with cover items (specifically statement coverage) that are excluded by pragma.

Excluding Objects from Coverage

You can exclude any number of lines or files, or condition and expression UDP truth table rows, from coverage statistics collection. The line exclusions can be instance-specific or they can apply to all instances in the enclosing design unit. Truth table row exclusions can be instance specific. You can also exclude nodes from toggle statistics collection using the toggle disable command.

The following methods can be used for excluding objects:

- Exclude Lines and Files Using the GUI
- Exclude Individual Metrics with Pragmas
- Exclude Lines and Rows from UDP Truth Tables
- Exclude Lines and Rows with the Coverage Exclude Command
- Exclude Nodes from Toggle Statistics
- Exclude Lines, Rows, Files and Toggle Nodes with a .do File

Exclude Lines and Files Using the GUI

There are several locations in the GUI where you can access commands to exclude lines or files:

- Right-click a file in Files tab of the Workspace pane and select **Code Coverage > Exclude Selected File** from the popup menu.
- Right-click an entry in the Main window Missed Coverage pane and select **Exclude Selection** or **Exclude Selection For Instance <inst_name>** from the popup menu.
- Right-click a line in the Hits column of the Source window and select **Exclude Coverage Line xxx**, **Exclude Coverage Line xxx For Instance <inst_name>**, or **Exclude Entire File**.

Exclude Individual Metrics with Pragmas

ModelSim also supports the use of source code pragmas to selectively turn coverage off and on for individual code coverage metrics. In Verilog, the pragmas are:

```
// coverage off
// coverage on
```

In VHDL, the pragmas are:

```
-- coverage off
-- coverage on
```

Bracket the line or lines you want to exclude with these pragmas.

Pragmas allow you to turn statement, branch, condition, expression and FSM coverage on and off independently. To create an exclusion, add an additional argument to the coverage on or coverage off pragma using characters to indicate the coverage metric. For example

```
// coverage off sce
```

turns off statement, condition and expression coverage in Verilog, and leaves the other metrics alone.

```
-- coverage on bf
```

turns on branch and FSM coverage in VHDL, leaving the other metrics alone.

Here are some points to keep in mind about using these pragmas:

- Pragmas are enforced at the design unit level only. For example, if you put "-- coverage off" before an architecture declaration, all statements in that architecture will be excluded from coverage; however, statements in all following design units will be included in statement coverage (until the next "-- coverage off").
- Pragmas cannot be used to exclude specific subconditions or subexpressions within lines, although they can be used for individual case statement alternatives.

Exclude Lines and Rows from UDP Truth Tables

You can also use exclusion filter files to exclude lines and rows from condition and expression UDP truth tables. Exclusions may be instance-specific.

Syntax

```
file_name -c | -e {<ln> <rn|rn1-rn2>...}
```

Arguments

- -c | -e

  Determines whether to exclude condition (-c) or expression (-e) UDP truth table rows.

- <ln> ...

  The line number containing the condition or expression. The line number and list of row numbers are surrounded by curly braces.

- <rn | rn1 - rn2>

  A space separated list of row numbers or ranges of row numbers referring to the UDP truth table rows that you want excluded.

Example

```
control.vhd
  72-76
  -c {78 1 3-6}
```

In this example, lines 72 through 76 will be excluded from code coverage in control.vhd file. Rows 1 and 3 through 6 in the condition truth table on line 78 will also be excluded.

Exclude Lines and Rows with the Coverage Exclude Command

You can use the coverage exclude command for direct exclusion of specific lines in a source file or rows within a condition or expression truth table.

Exclude Nodes from Toggle Statistics

To exclude nodes from toggle statistics collection, use the toggle disable command.

Exclude Lines, Rows, Files and Toggle Nodes with a .do File

In the case where you would like to exclude all types of coverage data (including toggle data) using a single file, the recommended method is to use the coverage exclude command in conjunction with the toggle disable command in a **.do** file. The **coverage exclude** command now offers all that is available in the exclusion file format and therefore alleviates the need to use an exclusion filter file for lines/rows/files and then separate **toggle disable** commands to exclude toggle data.

Saving and Recalling Exclusions

You may specify files and line numbers or condition and expression UDP truth table rows (see below for details) that you wish to exclude from coverage statistics. These exclusions will appear in the Current Exclusions pane. You can then create a *.do* file that contains these exclusions by making the Current Exclusions pane active and then choosing **File** > **Save** from the menus. To load this *.do* file during a future analysis, select the Current Exclusions pane and select **File** > **Open**.

Example of excluding, merging and reporting on several runs

Suppose you are doing a number of simulations, i, numbered from 1 to n.

1. Use vlib to create a working library.
2. Use vcom and/or vlog to compile.
3. Use vsim to load and run the design:

   ```
   vsim -c <design_i> -do "log -r /*; run -all; do <exclude_file_i.do>;
   coverage save <results_i>; quit -f"
   ```

   Note, you can have different exclude files <*exclude_file_i*> for each run i, numbered from 1 to n.
4. Use vcover merge to merge the coverage data:

   ```
   vcover merge <merged_results_file> <results_1> <results_2> ...
   <results_n>
   ```
5. Use vcover report to generate your report:

```
vcover report [switches_you_want] -output <report_file>
<merged_results_file>
```

Exclusions are invoked at the vsim step 3, so you can't undo any of the exclusions after your vsim runs.

All the various results files <*results_i*> contain the exclusion information inserted at step 3.

The exclusion information for the merged results file is derived by ORing the exclusion flags from each vsim run. So, for example, if runs 1 and 2 exclude xyz.vhd line 12, but the other runs don't exclude that line, the exclusion flag for xyz.vhd line 12 is set in the merged results since at least one of the runs excluded that line. Then the final **vcover report** will not show coverage results for file xyz.vhd line 12.

Let's suppose your <*exclude_file_i*> are all the same, and called *exclude.do*.

The contents of *exclude.do* file could be:

```
coverage exclude -srcfile xyz.vhd -linerange 12 55 67-90
coverage exclude -srcfile abc.vhd -linerange 3-6 9-14 77
coverage exclude -srcfile pqr.vhd -linerange all
```

This will exclude lines 12, 55, and 67 to 90 (inclusive) of file *xyz.vhd*; lines 3 to 6, 9 to 14, and 77 of *abc.vhd*; and all lines from *pqr.vhd*.

Default Filter File

The Tcl preference variable PrefCoverage(pref_InitFilterFrom) specifies a default filter file to read when a design is loaded with the **-coverage** switch. By default this variable is not set. See Simulator GUI Preferences for details on changing this variable.

Reporting Coverage Data

You have several options for creating reports on coverage data. To create reports when a simulation is loaded, use either the coverage report command, the toggle report command (see Toggle Coverage Reporting in this chapter), or the Coverage Report dialog.

To create reports when a simulation isn't loaded, use the vcover report command.

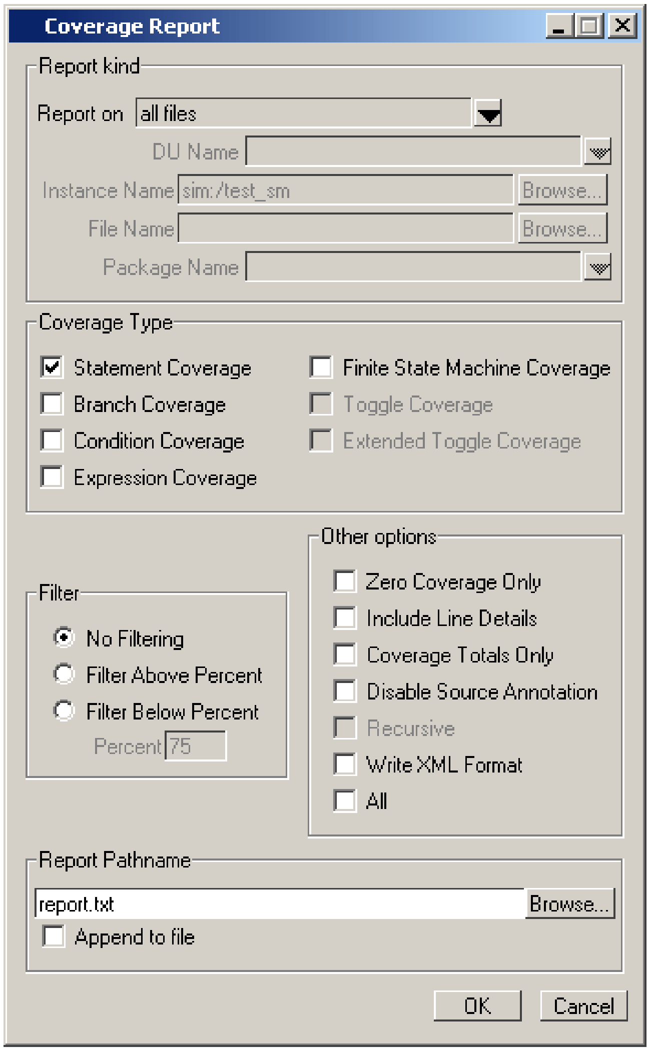
Example 14-1. Reporting Coverage Data from the Command Line

Here is a sample command sequence that outputs a code coverage report and saves the coverage data:

```
    vlog -cover bcesx ../rtl/host/top.v
    vlog -cover bcesx ../rtl/host/a.v
    vlog -cover bcesx ../rtl/host/b.v
    vlog -cover bcesx ../rtl/host/c.v

    vsim -c -coverage top
    run 1 ms
    coverage report -file d:\\sample\\coverage_rep
    coverage save d:\\sample\\coverage
```

Example 14-2. Reporting Coverage Data from the GUI

To access the Coverage Report dialog, right-click any object in the *Files* tab of the Workspace pane and select **Code Coverage** > **Code Coverage Reports;** or, select **Tools** > **Code Coverage** > **Reports**.



Report kind

Report on: all files

DU Name:

Instance Name: sim:/test_sm Browse...

File Name: Browse...

Package Name:

Coverage Type

☑ Statement Coverage
□ Branch Coverage
□ Condition Coverage
□ Expression Coverage
□ Finite State Machine Coverage
□ Toggle Coverage
□ Extended Toggle Coverage

Filter

● No Filtering
○ Filter Above Percent
○ Filter Below Percent
Percent: 75

Other options

□ Zero Coverage Only
□ Include Line Details
□ Coverage Totals Only
□ Disable Source Annotation
□ Recursive
□ Write XML Format
□ All

Report Pathname

report.txt Browse...

□ Append to file

Setting a Default Coverage Mode

You can specify a default coverage mode that persists from one ModelSim session to the next through the preference variable PrefCoverage(DefaultCoverageMode). The modes available allow you to specify that lists and data given in each report are listed by: file, design unit, or instance. By default, the report is listed by file. See Simulator GUI Preferences for details on changing this variable.

You may also specify a default coverage mode for the current invocation of ModelSim by using the -setdefault [byfile | byinstance | bydu] argument for either the coverage report or the vcover report command.

XML Output

You can output coverage reports in XML format by checking **Write XML Format** in the Coverage Report dialog or by using the **-xml** argument to the coverage report command.

The following example is an abbreviated "By Instance" report that includes line details:

```
<?xml version="1.0"?>
<report
lines="1"
byInstance="1">
<instance path="/test_delta/chip/control_126k_inst"
du="mode_two_control">
<source_table files="1">
<file fn="0" path="C:/modelsim_examples/coverage/Modetwo.v"></file>
</source_table>
<statements active="30" hits="17" percent="56.7"> </statements>
<statement_data>
<stmt fn="0" ln="39" st="1" hits="82"> </stmt>
<stmt fn="0" ln="42" st="1" hits="82"> </stmt>
<stmt fn="0" ln="44" st="1" hits="82"> </stmt>
```

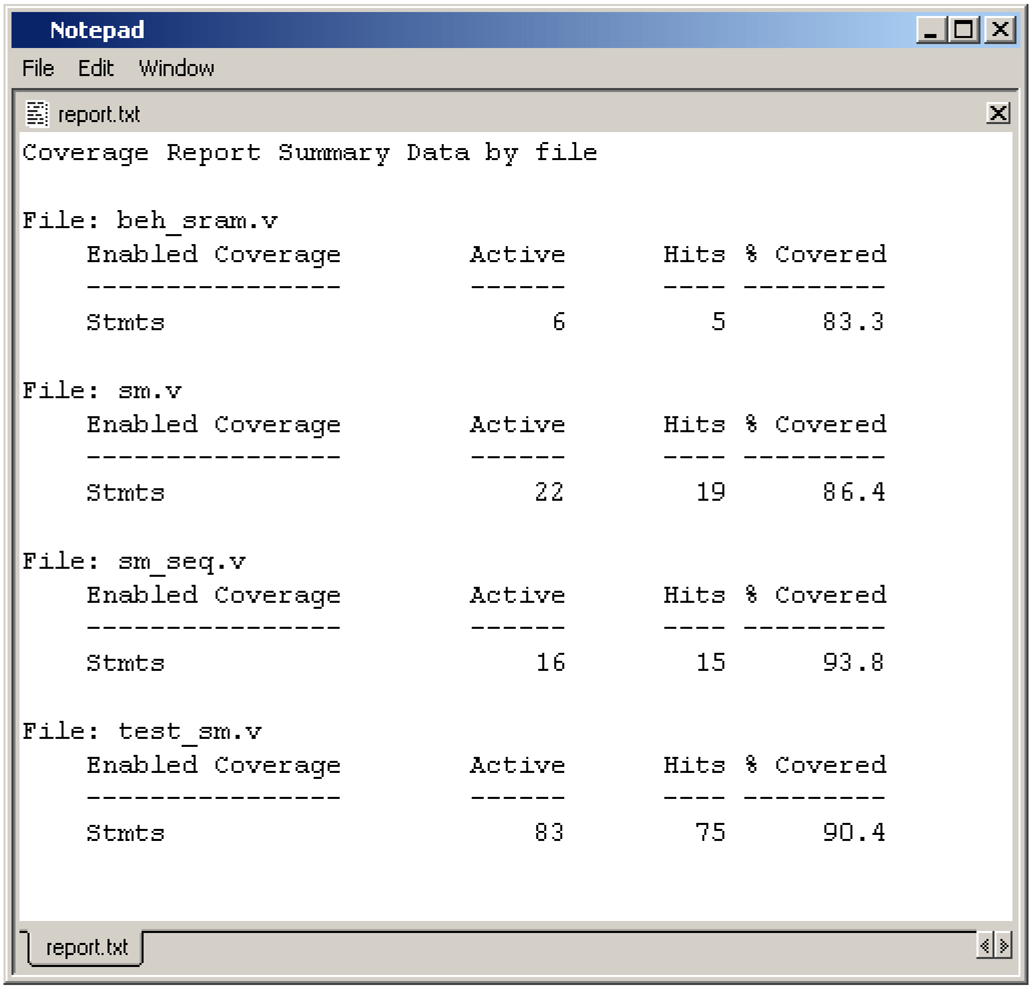
"fn" stands for filename, "ln" stands for line number, and "st" stands for statement.

There is also an XSL stylesheet named *covreport.xsl* located in
<install_dir>/examples/tutorials/vhdl/coverage, or
<install_dir>/examples/tutorials/verilog/coverage.
Use it as a foundation for building your own customized report translators.

Sample Reports

Below are abbreviated coverage reports with descriptions of select fields.

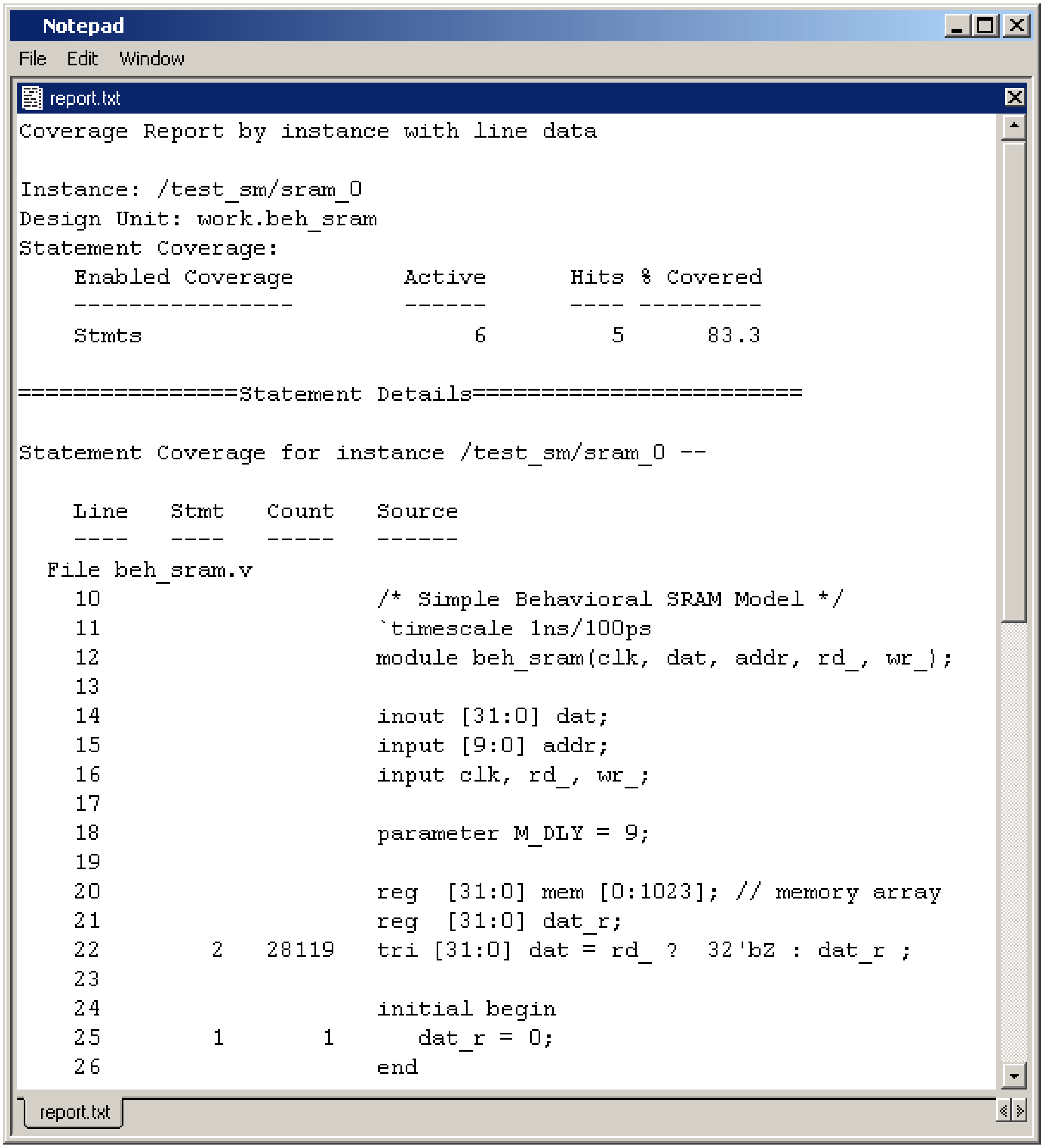
Statement Coverage Summary Report by File



```
Coverage Report Summary Data by file

File: beh_sram.v
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                        6         5      83.3

File: sm.v
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       22        19      86.4

File: sm_seq.v
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       16        15      93.8

File: test_sm.v
    Enabled Coverage        Active      Hits % Covered
    ----------------        ------      ---- ---------
    Stmts                       83        75      90.4
```

This report shows statement coverage by file with Active, Hits and % Covered columns. % Coverage shows statement hits divided by the number of active statements.

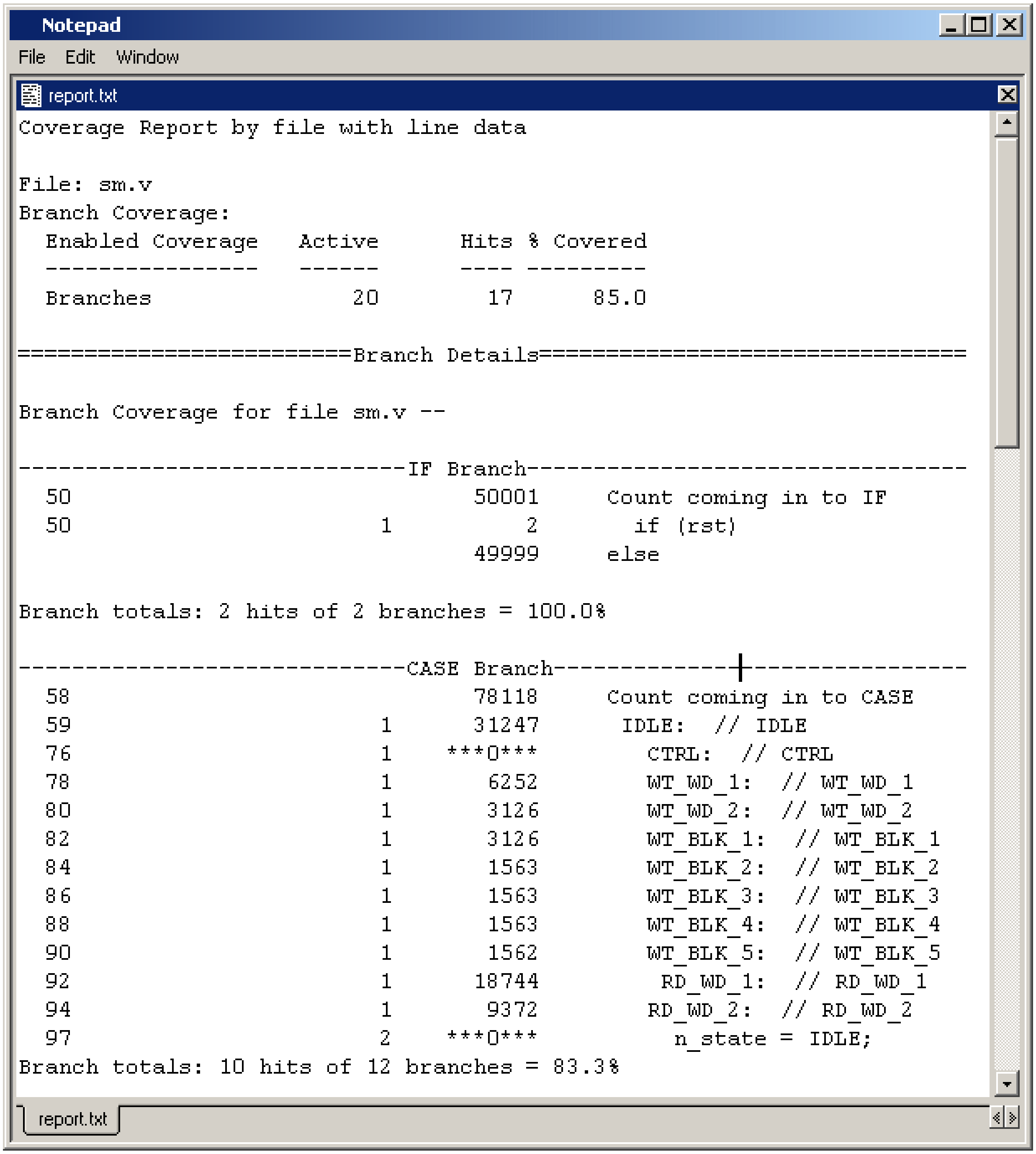
Instance Report with Line Details



```
Coverage Report by instance with line data

Instance: /test_sm/sram_0
Design Unit: work.beh_sram
Statement Coverage:
    Enabled Coverage            Active      Hits % Covered
    ----------------            ------      ---- ---------
    Stmts                            6         5      83.3

================Statement Details========================

Statement Coverage for instance /test_sm/sram_0 --

    Line   Stmt   Count   Source
    ----   ----   -----   ------
  File beh_sram.v
    10                    /* Simple Behavioral SRAM Model */
    11                    `timescale 1ns/100ps
    12                    module beh_sram(clk, dat, addr, rd_, wr_);
    13
    14                    inout [31:0] dat;
    15                    input [9:0] addr;
    16                    input clk, rd_, wr_;
    17
    18                    parameter M_DLY = 9;
    19
    20                    reg  [31:0] mem [0:1023]; // memory array
    21                    reg  [31:0] dat_r;
    22        2   28119   tri [31:0] dat = rd_ ?  32'bZ : dat_r ;
    23
    24                    initial begin
    25        1       1      dat_r = 0;
    26                    end
```

Each line may contain more than one statement, and each statement is given a number. The "Stmt" field identifies the statement number a specific line. The “Count” field shows the number of hits for the identified statement. So on line 22 of the report above, statement 2 received 28119 hits.

Branch Report



```
Coverage Report by file with line data

File: sm.v
Branch Coverage:
  Enabled Coverage   Active      Hits % Covered
  ----------------   ------      ---- ---------
  Branches               20        17      85.0

=========================Branch Details================================

Branch Coverage for file sm.v --

-----------------------------IF Branch---------------------------------
  50                              50001     Count coming in to IF
  50                       1          2       if (rst)
                                  49999     else

Branch totals: 2 hits of 2 branches = 100.0%

-----------------------------CASE Branch-------------------------------
  58                              78118     Count coming in to CASE
  59                       1      31247      IDLE:  // IDLE
  76                       1    ***0***        CTRL:  // CTRL
  78                       1       6252        WT_WD_1:  // WT_WD_1
  80                       1       3126        WT_WD_2:  // WT_WD_2
  82                       1       3126        WT_BLK_1:  // WT_BLK_1
  84                       1       1563        WT_BLK_2:  // WT_BLK_2
  86                       1       1563        WT_BLK_3:  // WT_BLK_3
  88                       1       1563        WT_BLK_4:  // WT_BLK_4
  90                       1       1562        WT_BLK_5:  // WT_BLK_5
  92                       1      18744         RD_WD_1:  // RD_WD_1
  94                       1       9372        RD_WD_2:  // RD_WD_2
  97                       2    ***0***          n_state = IDLE;
Branch totals: 10 hits of 12 branches = 83.3%
```

If an IF Branch ends in an "else" clause, the "else" count will be shown. Otherwise, an "All False" count will be given, which indicates how many times none of the conditions evaluated "true." If "INF" appears in the Count column, it indicates that the coverage count has exceeded ~4 billion (2^{32} -1). ’***0***’ indicates a zero count for that branch.

Coverage Statistics Details

This section describes how condition and expression coverage statistics are calculated.

Condition Coverage

Condition coverage analyzes the decision made in "if" and ternary statements and is an extension to branch coverage. A truth table is constructed for the condition expression and counts are kept for each row of the truth table that occurs. For example, the following IF statement:

```
Line 180: IF (a or b) THEN x := 0; else x := 1; endif;
```

reflects this truth table.

Table 14-4.

Truth table for line 180				
	counts	**a**	**b**	**(a or b)**
Row 1	5	1	-	1
Row 2	0	-	1	1
Row 3	8	0	0	0
unknown	0			

Row 1 indicates that (*a* or *b*) is true if *a* is true, no matter what *b* is. The "counts" column indicates that this combination has executed 5 times. The '-' character means "don't care." Likewise, row 2 indicates that the result is true if *b* is true no matter what *a* is, and this combination has executed zero times. Finally, row 3 indicates that the result is always zero when *a* is zero and *b* is zero, and that this combination has executed 8 times.

The unknown row indicates how many times the line was executed when one of the variables had an unknown state.

If more than one row matches the input, each matching row will be counted. If that is not the behavior you want and you would prefer no counts to be incremented on multiple matches, set "CoverCountAll=0" in your *modelsim.ini* file.

Values that are vectors are treated as subexpressions external to the table until they resolve to a boolean result. For example, take the IF statement:

Line 38:IF ((e = '1') AND (bus = "0111")) ...

A truth table will be generated in which bus = "0111" is evaluated as a subexpression and the result, which is boolean, becomes an input to the truth table. The truth table looks as follows:

Table 14-5.

Truth table for line 38				
	counts	**e**	**(bus="0111")**	**(e='1') AND (bus = "0111")**
Row 1	0	0	-	0
Row 2	10	-	0	0
Row 3	1	1	1	1
unknown	0			

Index expressions also serve as inputs to the table. Conditions containing function calls cannot be handled and will be ignored for condition coverage.

If a line contains a condition that is uncovered - some part of its truth table was not encountered - that line will appear in the Missed Coverage pane under the Conditions tab. When that line is selected, the condition truth table will appear in the Details pane and the line will be highlighted in the Source window.

Condition coverage truth tables are printed in coverage reports when the Condition Coverage type is selected in the Coverage Reports dialog (see Reporting Coverage Data), or when the **-lines** argument is specified in the coverage report command and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the **-dump** argument to the **coverage report** command.

ℹ **Tip**: Condition coverage does not do short circuit evaluation of IF conditions. So if you turn on condition coverage and you get a crash, that is the first thing you should check.

Expression Coverage

Expression coverage analyzes the expressions on the right hand side of assignment statements and counts when these expressions are executed. For expressions that involve logical operators, a truth table is constructed and counts are tabulated for conditions matching rows in the truth table.

For example, take the statement:

```
Line 236: x <= a xor (not b(0));
```

This statement results in the following truth table, with associated counts.

Table 14-6.

Truth table for line 236				
	counts	**a**	**b(0)**	**(a xor (not b(0)))**
Row 1	1	0	0	1
Row 2	0	0	1	0
Row 3	2	1	0	0
Row 4	0	1	1	1
unknown	0			

If a line contains an expression that is uncovered (some part of its truth table was not encountered) that line will appear in the Missed Coverage pane under the Expressions tab. When that line is selected, the expression truth table will appear in the Details pane and the line will be highlighted in the Source window.

As with condition coverage, expression coverage truth tables are printed in coverage reports when the Expression Coverage type is selected in the Coverage Reports dialog (see Reporting Coverage Data) or when the **-lines** argument is specified in the coverage report command and one or more of the rows has a zero hit count. To force the table to be printed even when it is 100% covered, use the **-dump** argument for the coverage report command.

Chapter 15
Finite State Machines

Because of the complexity of state machines, Finite State Machine (FSM) design is prone to the introduction of errors. It is important, therefore, to analyze the coverage of FSMs in RTL before going to the next stages of synthesis in the design cycle.

Types of FSM Coverage

In ModelSim, two kinds of coverage are defined for FSM's.

1. **State Coverage Metric:** This metric determines how many FSM states have been reached during simulation.
2. **Transition or FSM Arc Coverage Matrix:** This metric determines how many transitions have been exercised in the simulation of the state machine.

ModelSim also performs state reachability analysis on extracted FSMs and flags unreachable states.

All existing coverage command line options – such as coverage clear, coverage exclude, coverage report, etc. – can be used for FSM coverage.

Note

The functionality described in this chapter requires a coverage license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

FSM Recognition

FSM recognition happens as part of code generation. If, due to flow or overriding switches, code is generated in vopt (see Optimizing Designs with vopt), then FSMs are also recognized in vopt. If code is generated in vcom or vlog, then FSMs are recognized in vcom or vlog. But if FSM depends on any parameter/generics, then it is recognized only in vopt.

FSM recognition and coverage is enabled at compile time by using the **-cover f** or **-coverAll** arguments for vcom or vlog. Various design styles are used in Verilog and VHDL to specify FSMs. Their main features are as follows:

- There should be a finite number of states which the state variable can hold.
- The next state assignments to the state variable must be made under a clock.

- The next state value must depend on the current state value of the state variable.
- State assignments that do not depend on the current state value are considered reset assignments.

There are two types of supported FSMs:

1. using a single state variable
2. using a current state variable and a next state variable.

The following examples illustrate the two supported types.

Example 1 – Using a single state variable in Verilog

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;

parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
  if (rst) cst <= s0;
  else
    casex (cst)
       s0: begin out = inp[0]; if (enb) cst = s1; end
       s1: begin out = inp[1]; if (enb) cst = s2; end
       s2: begin out = inp[2]; if (enb) cst = s3; end
       s3: begin out = inp[3]; if (enb) cst = s4; end
       s4: begin out = inp[4]; if (enb) cst = s5; end
       s5: begin out = inp[4]; if (enb) cst = s6; end
       s6: begin out = inp[6]; if (enb) cst = s7; end
       s7: begin out = inp[7]; if (enb) cst = s0; end
       default: begin out = inp[5]; cst = s1; end
    endcase

endmodule
```

Example 2 – Using a single state variable in VHDL

```
RTL
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity fsm is port( in1 : in signed(1 downto 0);
                    in2 : in signed(1 downto 0);
                    en   : in std_logic_vector(1 downto 0);
                    clk: in std_logic;
                    reset : in std_logic_vector( 3 downto 0);
                    out1 : out signed(1 downto 0));
end fsm;
```

```
architecture arch of fsm is
  type my_enum is (s0 , s1, s2, s3,s4,s5);
  type mytype is array (3 downto 0) of std_logic;
  signal test : mytype;
  signal cst : my_enum;

  begin
    process(clk,reset)
    begin
       if(reset(1 downto 0) = "11") then
          cst <= s0;
       elsif(clk'event and clk = '1') then
          case cst is
              when s0 =>   cst <= s1;
              when s1 =>   cst <= s2;
              when s2 =>   cst <= s3;
              when others =>   cst <= s0;
          end case;
      end if;
    end process;
end arch;
```

Example 3 – Using a current state variable and a single next state variable in Verilog

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;
parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
  if (rst) cst <= s0;
  else     cst <= nst;

always @(cst or inp or enb)
begin
  casex (cst)
  s0: begin out = inp[0]; if (enb) nst = s1; end
  s1: begin out = inp[1]; if (enb) nst = s2; end
  s2: begin out = inp[2]; if (enb) nst = s3; end
  s3: begin out = inp[3]; if (enb) nst = s4; end
  s4: begin out = inp[4]; if (enb) nst = s5; end
  s5: begin out = inp[4]; if (enb) nst = s6; end
  s6: begin out = inp[6]; if (enb) nst = s7; end
  s7: begin out = inp[7]; if (enb) nst = s0; end
  default: begin out = inp[5]; nst = s1; end
  endcase
end
endmodule
```

Example 4 – Using a current state variable and a single next state variable in VHDL

```
RTL
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.pack.all;
entity fsm is port( in1 : in signed(1 downto 0);
                     in2 : in signed(1 downto 0);
                     en  : in std_logic_vector(1 downto 0);
                     clk: in std_logic;
                     reset : in std_logic_vector( 3 downto 0);
                     out1 : out signed(1 downto 0));
end fsm;

architecture arch of fsm is
  type my_enum is (s0 , s1, s2, s3,s4,s5);
  type mytype is array (3 downto 0) of std_logic;
  signal test : mytype;
  signal cst, nst : my_enum;

  begin
    process(clk,reset)
    begin
       if(reset(1 downto 0) = "11") then
          cst <= s0;
       elsif(clk'event and clk = '1') then
         cst <= nst;
      end if;
   end process;

   process(cst)
   begin
          case cst is
              when s0 =>   nst <= s1;
              when s1 =>   nst <= s2;
              when s2 =>   nst <= s3;
              when others =>   nst <= s0;
          end case;
    end process;
end arch;
```

Note

Multi-level next state assignments are not supported.

Supported Design Styles for FSM Extraction

The following styles of writing FSMs in VHDL and Verilog are supported:

- Using enum or constant literals or constant expressions and Case statements
- Using single If-Else statement — FSMs are extracted from If-Else statements in Verilog and VHDL only when FSM transitions are specified under equality comparisons with

the current state variable. For example, only comparisons like (cst == 3’b000) in Verilog and (cst = “000”) in VHDL are supported. Any other comparisons such as <, >, <=. !=, and /= are not supported.

- Using constants — In Verilog, constants can be specified using localparams or parameters. Parameters are considered constant only in the Vopt flow. Hence, if FSMs are written with parameters as state values, they will be recognized only in the Vopt flow.
- Using wait/select statement (VHDL only)
- Using current state variable
- Using current state and next state variable
- Using basic integral SystemVerilog types

Unsupported Design Styles

The following design styles of writing FSMs in VHDL and Verilog are not supported:

- Complex types - Advanced SystemVerilog types — In VHDL, state variables as recorded fields
- Using bit-selects or part-selects (Verilog) and index expressions or slice expressions (VHDL)
- FSMs in Generate statements — In VHDL, FSMs are not extracted in generate statements. In Verilog, FSMs are extracted for generate statements only if the for-generate statements have been unrolled or if-generate statements have been converted to gen-blocks after resolving them.
- Using **when-else** logic (VHDL only)
- Multiple **case** statements or nested **case** statements used to define FSM transitions
- Multiple **if** statements used to define FSM transitions
- Mixed **case** and **if-else** statements used to specify FSM transitions
- One-Hot FSMs with assignments to bit-positions

FSM extraction reporting

Output messages are generated to report the FSMs extracted from the design. These reports contain the following information.

- The current state variable
- The next state variable (if any)

- The clock under which the state variable is assigned
- The state set of the FSM
- The reset states of the FSM
- The set of state transitions along with the line numbers from which they have been identified

State reachability analysis is carried out on the extracted FSMs and the unreachable states, if any, are also reported. A state is marked as unreachable if it cannot be reached from the reset states. These messages can be suppressed using the "-suppress 1947" arguments for vcom or vlog. The following Verilog example will illustrate the above points.

Verilog reporting example

```
RTL
module fsm_test (out, inp, clk, enb, cnd, rst);
output out; reg out;
input [7:0] inp;
input clk, enb, cnd, rst;

parameter [2:0] s0 = 3'h0, s1 = 3'h1, s2 = 3'h2, s3 = 3'h3,
                s4 = 3'h4, s5 = 3'h5, s6 = 3'h6, s7 = 3'h7;
reg [2:0] cst, nst;

always @(posedge clk or posedge rst)
  if (rst) cst <= s0;
  else     cst <= nst;

always @(cst or inp or enb)
begin
  casex (cst)
  s0: begin out = inp[0]; if (enb) nst = s1; end
  s1: begin out = inp[1]; if (enb) nst = s2; end
  s2: begin out = inp[2]; if (enb) nst = s4; end
  s3: begin out = inp[3]; if (enb) nst = s6; end
  s4: begin out = inp[4]; if (enb) nst = s3; end
  s5: begin out = inp[4]; if (enb) nst = s6; end // state s5 is an
unreachable state
  s6: begin out = inp[6]; if (enb) nst = s7; end
  s7: begin out = inp[7]; if (enb) nst = s0; end
  default: begin out = inp[5]; nst = s1; end
  endcase
end

endmodule

COMPILE REPORT
** Note: (vlog-1947)   FSM RECOGNITION INFO
        Fsm detected in : fsm-unreach.v
        Current State Variable : cst : fsm-unreach.v(8)
        Next State Variable : nst : fsm-unreach.v(8)
        Clock : clk
        Reset States are: { s0 }
        State Set is : { s0 , s1 , s2 , s4 , s3 , s6 , s5 , s7 }
```

```
Transition table is
        -------------------------------------------
        s0   =>   s1            Line : (18 => 18)
        s0   =>   s0            Line : (12 => 12)
        s1   =>   s2            Line : (19 => 19)
        s1   =>   s0            Line : (12 => 12)
        s2   =>   s4            Line : (20 => 20)
        s2   =>   s0            Line : (12 => 12)
        s4   =>   s3            Line : (22 => 22)
        s4   =>   s0            Line : (12 => 12)
        s3   =>   s6            Line : (21 => 21)
        s3   =>   s0            Line : (12 => 12)
        s6   =>   s7            Line : (24 => 24)
        s6   =>   s0            Line : (12 => 12)
        s5   =>   s6            Line : (23 => 23)
        s5   =>   s0            Line : (12 => 12)
        s7   =>   s0            Line : (25 => 25) (12 => 12)
        -------------------------------------------
INFO : State s5 is unreachable from the reset states
```

Disabling FSM Extraction Using Pragmas

The user can also disable FSM extraction using pragmas. The "coverage fsm_off <cst_var_name>" pragma turns off FSM recognition at compile time for the FSM whose current state variable name has been specified in the pragma. This pragma must be written after the declaration of the current state variable.

VHDL example

```
RTL
entity fsm1 is
  port(clk : bit;
       reset : bit);
end fsm1;

architecture arch of fsm1 is
type state_codes is (s0, s1, s2, s3);
begin
   process
   variable curr_state : integer range 0 to 3;
   variable next_state : integer range 0 to 3;

   -- coverage fsm_off curr_state

   begin
       wait until clk'event and clk = '1';
       curr_state := next_state;
      if(reset = '1') then
         curr_state := 0;
       else
         case curr_state is
         when 0 => next_state := 1;
         when 1 => next_state := 2;
         when 2 => next_state := 3;
         when 3 => next_state := 0;
```

```
        when others => next_state := 0;
             end case;
           end if;
       end process;
   end arch;

   COMPILE REPORT
   ** Warning: [13] fsm-pragma.vhdl(14): Detected coverage fsm_off pragma:
   Turning off FSM coverage for "curr_state".
```

Viewing FSM coverage in the GUI

Once the FSMs have been detected, you can simulate the design and view coverage results in the graphic user interface.

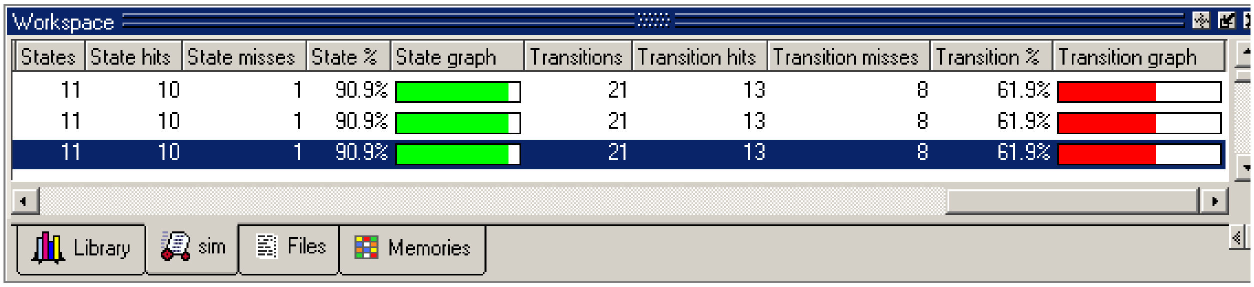
1. Compile the design with the **-cover f** argument for the vcom or vlog command.
2. Load the design with the **-coverage** argument for the vsim command.
3. Run the simulation.

FSM coverage data will appear in the Workspace, Objects, Missed Coverage, Instance Coverage, Details, and Current Exclusions window panes. In addition, you can generate a graphical state machine diagram in the FSM Viewer by double-clicking any FSM item in the Missed Coverage pane.

Workspace

Both the **sim** and the **Files** tab of the Workspace will display FSM coverage data in the following columns: States, State hits, State misses, State graph, Transitions, Transition hits, Transition misses, Transition %, and Transition graphs.

Figure 15-1. FSM coverage data in the Workspace



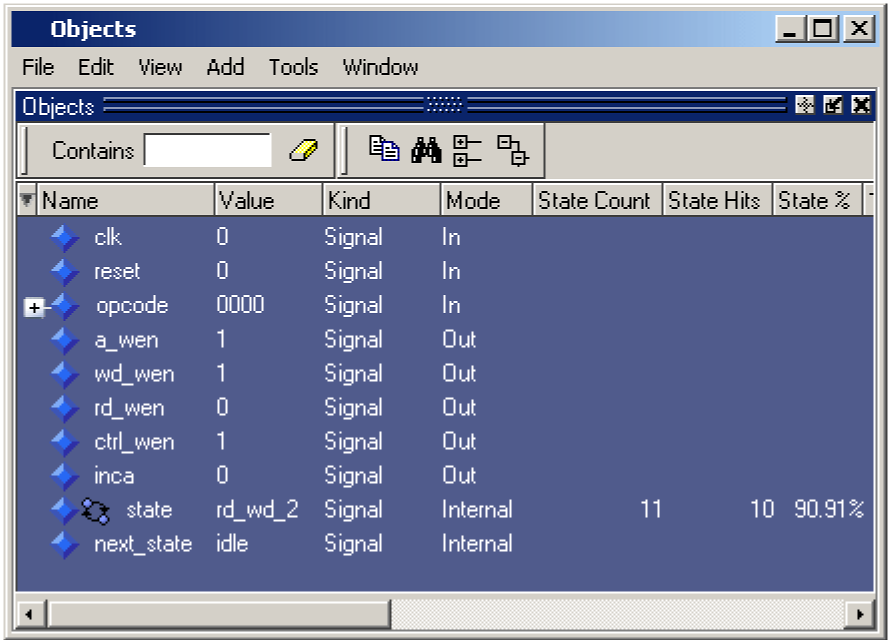
States	State hits	State misses	State %	State graph	Transitions	Transition hits	Transition misses	Transition %	Transition graph
11	10	1	90.9%	🟩	21	13	8	61.9%	🟥
11	10	1	90.9%	🟩	21	13	8	61.9%	🟥
11	10	1	90.9%	🟩	21	13	8	61.9%	🟥

The State % is the State hits divided by the States. Likewise, Transition % is the Transition hits divided by the Transitions. The State graph and the Transition graph will display a green bar with State % or Transition %, respectively, is 90% or greater. The bar graphs will be red for percentages under 90.

Objects

The Objects pane will display FSM coverage data in the State Count, State Hits and State % columns (Figure 15-2). The icon that appears next to the *state* variable name is also a button . Clicking this button will open a state machine view in the MDI area (see FSM Viewer).

Figure 15-2. FSM coverage in the Objects pane



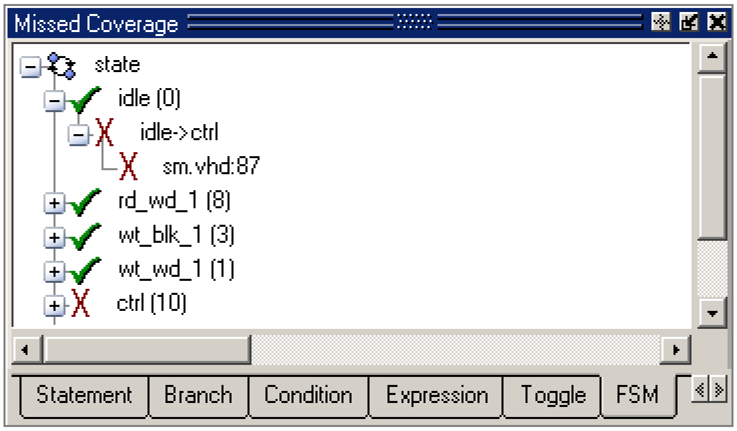
Name	Value	Kind	Mode	State Count	State Hits	State %
clk	0	Signal	In			
reset	0	Signal	In			
opcode	0000	Signal	In			
a_wen	1	Signal	Out			
wd_wen	1	Signal	Out			
rd_wen	0	Signal	Out			
ctrl_wen	1	Signal	Out			
inca	0	Signal	Out			
state	rd_wd_2	Signal	Internal	11	10	90.91%
next_state	idle	Signal	Internal			

Missed Coverage

The Missed Coverage pane contains an FSM tab that lists states and transitions that have been fully covered during simulation. Transitions are listed under states, and the source line numbers for each transition is listed under its respective transition. A design unit for file that contains a state machine must be selected in the workspace before anything will appear here. In the Missed Coverage pane, the icon that appears next to the *state* variable name is also a button . Pressing this button will open a state machine view in the MDI area (see FSM Viewer).

When you select a state, the state name will be highlighted in the FSM Viewer. When you select a transition, that transition line will be highlighted in the FSM Viewer. When you select the source line number for the transition, a Source view will open in the MDI frame and display the line number you have selected.

Figure 15-3. FSM Missed Coverage

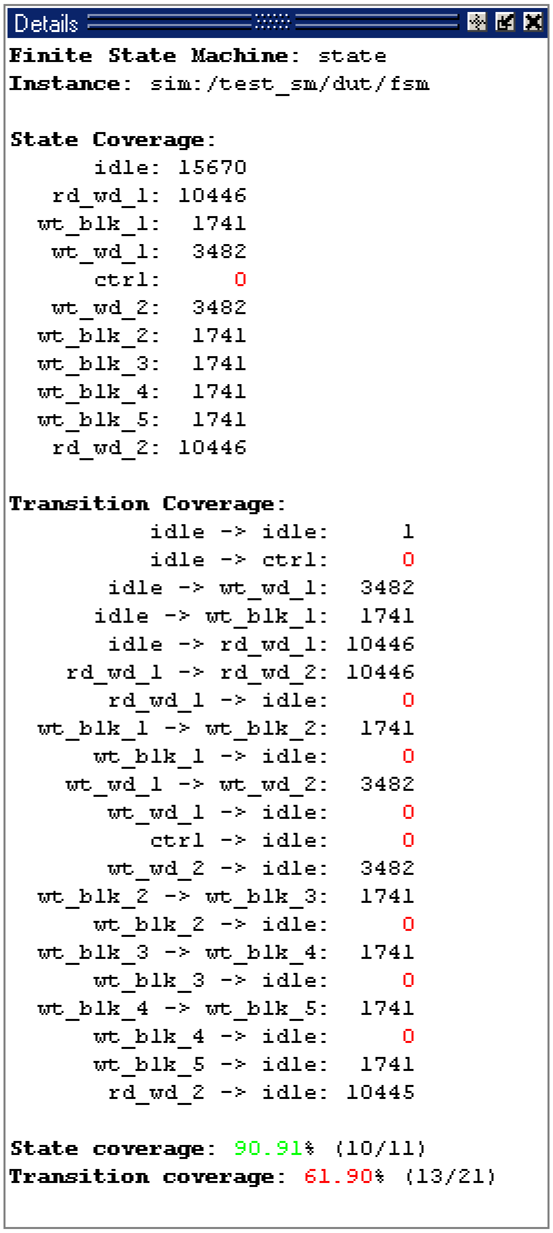


```
state
  ✓ idle (0)
    X idle->ctrl
      X sm.vhd:87
  ✓ rd_wd_1 (8)
  ✓ wt_blk_1 (3)
  ✓ wt_wd_1 (1)
  X ctrl (10)
```

Details

Select any FSM item in the Missed Coverage pane to show the details of state and transition coverage in the Details pane.

Figure 15-4. FSM Details



```
Finite State Machine: state
Instance: sim:/test_sm/dut/fsm

State Coverage:
      idle: 15670
   rd_wd_1: 10446
  wt_blk_1:  1741
   wt_wd_1:  3482
      ctrl:     0
   wt_wd_2:  3482
  wt_blk_2:  1741
  wt_blk_3:  1741
  wt_blk_4:  1741
  wt_blk_5:  1741
   rd_wd_2: 10446

Transition Coverage:
          idle -> idle:     1
          idle -> ctrl:     0
       idle -> wt_wd_1:  3482
      idle -> wt_blk_1:  1741
       idle -> rd_wd_1: 10446
    rd_wd_1 -> rd_wd_2: 10446
       rd_wd_1 -> idle:     0
  wt_blk_1 -> wt_blk_2:  1741
      wt_blk_1 -> idle:     0
    wt_wd_1 -> wt_wd_2:  3482
       wt_wd_1 -> idle:     0
          ctrl -> idle:     0
       wt_wd_2 -> idle:  3482
  wt_blk_2 -> wt_blk_3:  1741
      wt_blk_2 -> idle:     0
  wt_blk_3 -> wt_blk_4:  1741
      wt_blk_3 -> idle:     0
  wt_blk_4 -> wt_blk_5:  1741
      wt_blk_4 -> idle:     0
      wt_blk_5 -> idle:  1741
       rd_wd_2 -> idle: 10445

State coverage: 90.91% (10/11)
Transition coverage: 61.90% (13/21)
```

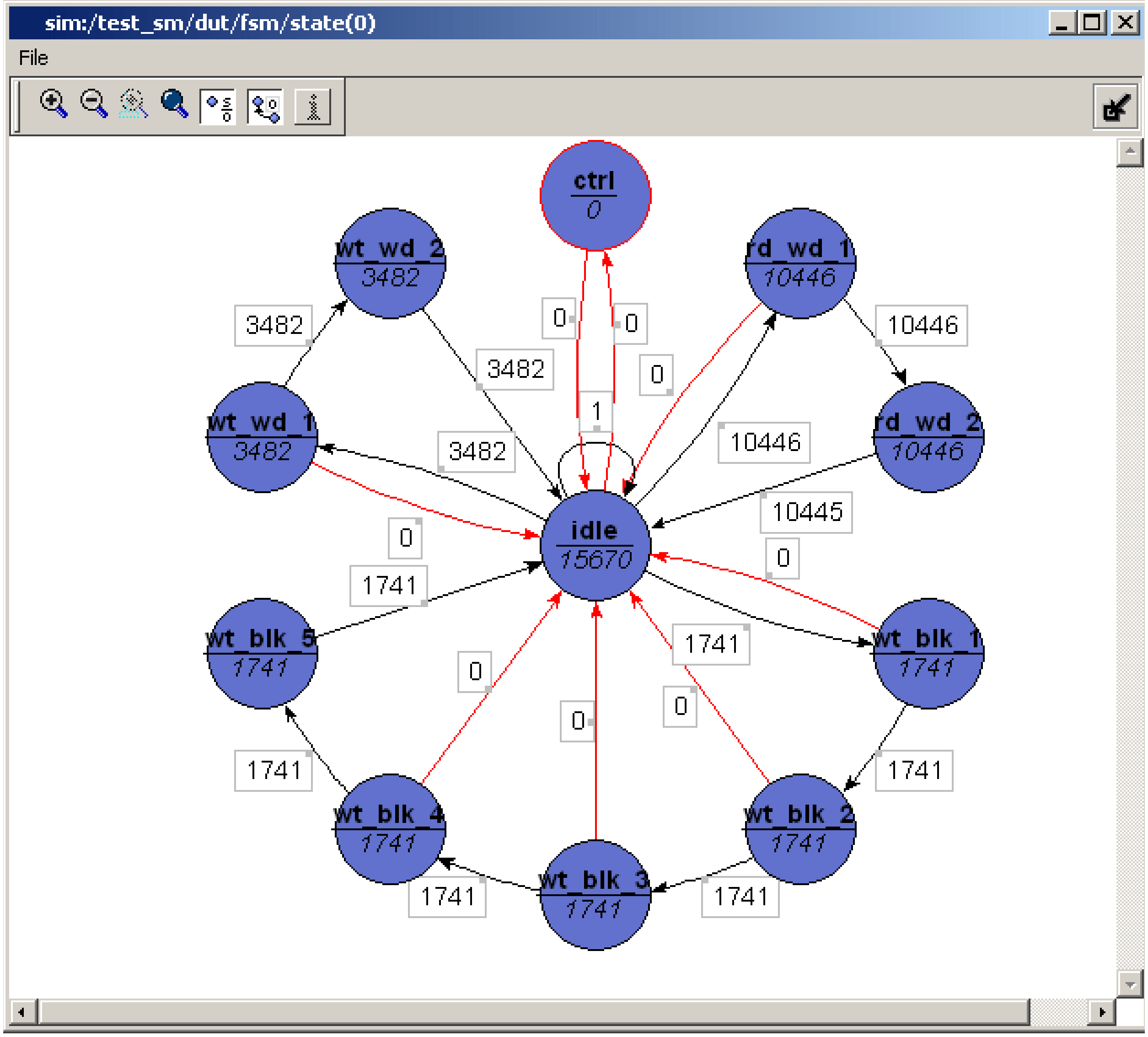
Instance Coverage

The Instance Coverage pane will display FSM coverage data in the following columns: States, State hits, State misses, State graph, Transitions, Transition hits, Transition misses, Transition %, and Transition graphs.

FSM Viewer

The icon that appears next to the state variable name in the Objects, Locals or Missed Coverage panes is also a button ⚙. Pressing this button will open a state machine view in the MDI area. This is the FSM Viewer. The numbers in boxes are the state and transition hit counts.

Figure 15-5. The FSM Viewer



The FSM Viewer is dynamically linked to the Details pane. If you select a bubble or a line in this diagram, the Details pane will display all FSM details, as shown in Figure 15-4. If you select a bubble, its name will be bold-faced. If you select a line, the names of both bubbles connected to the line will be bold-faced.

Using the Mouse in the FSM Viewer

These mouse operations are defined for the FSM Viewer:

- The mouse wheel performs zoom & center operations on the diagram. Mouse up will zoom out. Mouse down will zoom in. Whether zooming in or out, the view will recenter towards the mouse location.
- LMB (button-1) click will select the item under the mouse.
- Shift-LMB (button-1) click will extend the current selection.
- LMB (button-1) drag will select items inside the bounding box.
- MMB (button-3) drag has similar definitions as found in the Dataflow and Wave windows:
    - o Drag up and to the left will Zoom Full.
    - o Drag up and to the right will Zoom Out. The amount is determined by the distance dragged.
    - o Drag down and to the left will Zoom Selected.
    - o Drag down and to the right will Zoom In on the area of the bounding box.

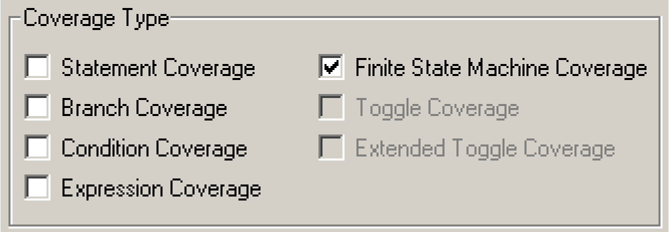
The Zoom toolbar has 3 additional mode buttons.

- The Show State Counts button will turn on/off the display of the state hit count.
- The Show Transition Counts button will turn on/off the display of the transition hit count.
- The Info Mode button **i** turns on the hover display mode for either or both hit counts, whichever are currently not displayed.

FSM Coverage Reports

You can initiate FSM coverage reports using the GUI or by entering coverage report commands at the command line. Using the GUI, select **Tools** > **Code Coverage** > **Reports** to open the Coverage Report dialog. In the Coverage Type section of the dialog, select Finite State Machine Coverage.

Figure 15-6. Coverage Type section of Coverage Report dialog



Coverage Type

☐ Statement Coverage
☐ Branch Coverage
☐ Condition Coverage
☐ Expression Coverage
☑ Finite State Machine Coverage
☐ Toggle Coverage
☐ Extended Toggle Coverage

The various coverage reporting commands available to view FSM coverage reports are detailed below.

Coverage Summary by Instance

Data is collected for all FSMs in each instance, merged together and reported when the following command is used:

```
coverage report -select f -byInst
```

The format of the FSM coverage summary by instance report is as follows:

```
# Coverage Report Summary Data by Instance
#
# FSM Coverage:
#
# File     Num_FSM  States  Hits    %    Trasitions  Hits    %
# ----     -------  ------  ----  -----  ----------  ----  -----
# /top/t1     5       26     13   50.0       50       20   40.0
```

Coverage Summary by Design Unit

Data is collected for all FSMs in all instances of each design unit. This data is then merged and reported when the following command is used:

```
coverage report -select f -byDu
```

The format of the FSM coverage summary by design unit report is as follows:

```
# Coverage Report Summary Data by Design Unit
#
# FSM Coverage:
#
# File     Num_FSM  States  Hits    %    Trasitions  Hits    %
# ----     -------  ------  ----  -----  ----------  ----  -----
# ENT(ARCH)   5       26     13   50.0       50       20   40.0
```

Coverage Summary by File

Data is collected for FSMs for all design units defined in each file and given in the report when the following command is used:

```
coverage report -select f -byFile
```

The format of the coverage summary by file is as follows:

```
# Coverage Report Summary Data by file
#
# FSM Coverage:
```

```
#
# File       Num_FSM  States  Hits    %    Trasitions  Hits    %
# ----       -------  ------  ----  -----  ----------  ----  -----
# bus_arb.vhd   5       26     13   50.0       50       20   40.0
```

Coverage Details by Instance

Cverage details for each FSM can also be reported by file, instance or design unit by appending the "-lines" option to the above commands. These reports specify the states, and transitions of the FSMs along with the number of times they have been hit in the simulation. Those states and transitions which have not been hit are listed separately. The command used is:

```
coverage report -select f -byInst -lines
```

The format of a coverage details report by instance, with line data, is as follows:

```
# Coverage Report by instance with line data
#
# FSM Coverage:
#
#          Inst                                  DU
# ----------------------------     -----------------------------
# /fsm_test_vhdl_10_config_rtl      fsm_test_vhdl_10_config_rtl
#
# Num_Fsm    States    Hits    %     Transitions    Hits    %
# -------    ------    ----  -----   -----------    ----  -----
#   1          6        6    100.0       17          6     35.3
#
# =============================FSM Details=============================
#
# FSM Coverage for instance /fsm_test_vhdl_10_config_rtl --
#
# FSM_ID [0]
#      Current State Object : state
#      ----------------------
#      State Value MapInfo :
#      ---------------------
#               State Name            Value
#               ----------            -----
#                     st0             0000
#                     st1             0001
#                     st2             0010
#                     st3             0011
#                     st4             0100
#                     st5             1111
#      Covered States :
#      ----------------
#                    State        Hit_count
#                    -----        ---------
#                     st0              1
#                     st1              1
#                     st2              1
#                     st3              2
#                     st4              1
#                     st5                2
#      Covered Transitions :
#      ---------------------
#                 Trans_ID   Transition         Hit_count
#                 --------   ----------         ---------
#                        4    st1 -> st2                1
#                        7    st2 -> st3                1
#                        9    st3 -> st3                1
#                       10    st3 -> st4                1
#                       13    st4 -> st0                1
#                       15    st5 -> st5                1
#      Uncovered Transitions :
#      -----------------------
#                 Trans_ID   Transition
#                 --------   ----------
#                        0    st0 -> st0
#                        1    st0 -> st1
#                        2    st0 -> st5
#                        3    st1 -> st1
#                        5    st1 -> st5
#                        6    st2 -> st2
#                        8    st2 -> st5
#                       11    st3 -> st5
#                       12    st4 -> st4
#                       14    st4 -> st5
#                       16    st5 -> st0
#
# Fsm_id   States    Hits    %     Transitions    Hits    %
# -------  ------    ----  -----   -----------    ----  -----
#    0       6        6    100.0       17          6    35.3
```

Coverage Details by Design Unit

The command used for reporting FSM coverage details by design unit is as follows:

```
coverage report -select f -byDu -lines
```

The format of a coverage details report by design unit will look like the following:

```
# Coverage Report by DU with line data
#
# FSM Coverage:
#
# Inst              DU
# ----  -----------------------------
#  --    fsm_test_vhdl_10_config_rtl
#
# Num_Fsm   States    Hits    %     Transitions    Hits    %
# -------   ------    ----  -----   -----------    ----  -----
#    1        6        6    100.0       17          6    35.3
#
#
# =============================FSM Details=============================
#
# FSM Coverage for Design Unit fsm_test_vhdl_10_config_rtl --
```

```
#
# FSM_ID [0]
#      Current State Object : state
#      ----------------------
#      State Value MapInfo :
#      ---------------------
#               State Name            Value
#               ----------            -----
#                     st0             0000
#                     st1             0001
#                     st2             0010
#                     st3             0011
#                     st4             0100
#                     st5             1111
#      Covered States :
#      ----------------
#                    State        Hit_count
#                    -----        ---------
#                     st0                1
#                     st1                1
#                     st2                1
#                     st3                2
#                     st4                1
#                     st5                2
#      Covered Transitions :
#      ---------------------
#                 Trans_ID   Transition        Hit_count
#                 --------   ---------- ---------
#                        4    st1 -> st2 1
#                        7    st2 -> st3 1
#                        9    st3 -> st3 1
#                       10    st3 -> st4 1
#                       13    st4 -> st0 1
#                       15    st5 -> st5 1
#      Uncovered Transitions :
#      -----------------------
#                 Trans_ID   Transition
#                 --------   ----------
#                        0    st0 -> st0
#                        1    st0 -> st1
#                        2    st0 -> st5
#                        3    st1 -> st1
#                        5    st1 -> st5
#                        6    st2 -> st2
#                        8    st2 -> st5
#                       11    st3 -> st5
#                       12    st4 -> st4
#                       14    st4 -> st5
#                       16    st5 -> st0
# Inst            DU
# ---- -----------------------------
#  --   fsm_test_vhdl_10_config_rtl
#
# Fsm_id   States    Hits    %     Transitions    Hits    %
# -------  ------    ----  -----   -----------    ----  -----
#    0       6        6    100.0        17          6    35.3
```

Coverage Details by File

The command used for reporting FSM coverage details by file is:

```
coverage report -select f -byFile -lines
```

The format used to report FSM coverage details by file will look like the following:

```
# Coverage Report by file with line data
#
# FSM Coverage:
#
#  File
# --------------------------------
# config_rtl_fsm_test_vhdl_10.vhdl
#
# Num_Fsm   States    Hits    %     Transitions    Hits    %
# -------   ------    ----  -----   -----------    ----  -----
#    1        6        6    100.0        17          6    35.3
#
# =============================FSM Details=============================
#
# FSM Coverage for file src/vhdl/fsm7/config_rtl_fsm_test_vhdl_10.vhdl --
#
#
# FSM_ID [0]
#      Current State Object : state
#      ----------------------
#      State Value MapInfo :
#      ---------------------
#               State Name            Value
#               ----------            -----
#                     st0             0000
#                     st1             0001
#                     st2             0010
#                     st3             0011
#                     st4             0100
#                     st5             1111
#      Covered States :
#      ----------------
#                    State        Hit_count
#                    -----        ---------
#                     st0                1
#                     st1                1
#                     st2                1
#                     st3                2
#                     st4                1
#                     st5                2
#      Covered Transitions :
#      ---------------------
#                 Trans_ID   Transition        Hit_count
#                 --------   ----------        ---------
#                        4    st1 -> st2                1
#                        7    st2 -> st3                1
#                        9    st3 -> st3                1
#                       10    st3 -> st4                1
#                       13    st4 -> st0                1
#                       15    st5 -> st5                1
#      Uncovered Transitions :
#      -----------------------
#                 Trans_ID   Transition
#                 --------   ----------
#                        0    st0 -> st0
#                        1    st0 -> st1
#                        2    st0 -> st5
#                        3    st1 -> st1
#                        5    st1 -> st5
#                        6    st2 -> st2
#                        8    st2 -> st5
#                       11    st3 -> st5
#                       12    st4 -> st4
#                       14    st4 -> st5
#                       16    st5 -> st0
#              File                                 DU
# --------------------------------   ---------------------------------
# config_rtl_fsm_test_vhdl_10.vhdl   vhdl  fsm_test_vhdl_10_config_rtl
#
# Fsm_id   States    Hits    %     Transitions    Hits    %
# -------- ------    ----  -----   -----------    ----  -----
#    0       6        6    100.0        17          6    35.3
```

Coverage Using the -zeros Option

When the -zeros option is given with the coverage report command, only the uncovered states and transitions will be printed. Coverage report will list the missed count and percentage.

The format of the FSM coverage report using the -zeros option will appear as follows:

```
# Coverage Zeros Report Summary by Design Unit
#

# FSM Coverage:
# Inst              DU
# ----  -----------------------------
#  --    fsm_test_vhdl_10_config_rtl
#
# Num_Fsm   States    Misses    %     Transitions   Misses    %
# -------   ------    ------  -----   -----------   ------  -----
#    1        6          0     0.0        17          11    64.7
#
# Coverage Report by instance of lines with zero counts
#
# FSM Coverage:
#           Inst                              DU
# ----------------------------   ---------------------------
# /fsm_test_vhdl_10_config_rtl   fsm_test_vhdl_10_config_rtl
#
# Num_Fsm  States   Misses    %    Transitions   Misses      %
# -------  ------   ------  -----  -----------   ------    -----
#    1        6       0      0.0        17         11       64.7
#
# Coverage Report by file of statements with zero counts (non instance
specific)
```

```
#
# FSM Coverage:
#                     File
# ----------------------------------------------
# src/vhdl/fsm7/config_rtl_fsm_test_vhdl_10.vhdl
#
# Num_Fsm   States   Misses     %    Transitions   Misses     %
# --------  ------   ------   -----  -----------   ------   -----
#     1       6        0       0.0        7          11      64.7
```

Coverage Using the -above, -below Options

Reports will be generated for those states and transitions whose coverage percentages fall within the given limits defined by the -above and -below options for the coverage report command.

The format of the report will be as follows:

```
# Coverage Report by file, with line data, having coverage > 10.0 percent
#
# FSM Coverage:
#               File
# --------------------------------
# config_rtl_fsm_test_vhdl_10.vhdl
#
# Num_Fsm   States   Hits     %     Transitions   Hits     %
# --------  ------   ----   -----   -----------   ----   -----
#    1         6       6    100.0        17         6     35.3
```

Coverage Using the -totals Option

Appending the -totals option to the coverage report command will give a summary of the all the states and transitions along with their hits and percentage coverages.

The format of the report will be as follows:

```
# Coverage Report Totals BY FILES: Number of Files 0,
#   Active Statements 0, Hits 0, Statement Coverage Percentage 100.0
#   Active Branches 0, Hits 0, Branch Coverage Percentage 100.0
#   Active Conditions 0, Hits 0, Condition Coverage Percentage 100.0
#   Active Expressions 0, Hits 0, Expression Coverage Percentage 100.0
#   Active States 6, Hits 6, State Coverage Percentage 100.0
#   Active Transitions 17, Hits 6, Transition Coverage Percentage 35.3
#
# Coverage Report Totals BY INSTANCES: Number of Instances 1
#   Active Statements 0, Hits 0, Statement Coverage Percentage 100.0
#   Active Branches 0, Hits 0, Branch Coverage Percentage 100.0
#   Active Conditions 0, Hits 0, Condition Coverage Percentage 100.0
#   Active Expressions 0, Hits 0, Expression Coverage Percentage 100.0
#   Active States 6, Hits 6, State Coverage Percentage 100.0
#   Active Transitions 17, Hits 6, Transition Coverage Percentage 35.3
```

FSM Coverage Exclusions

In ModelSim, any transition of an FSM can be excluded from the coverage reports using the coverage exclude or coverage clear commands at the command line.

Using the coverage exclude command

The coverage exclude command is used to exclude the specified transitions from coverage reports. For example:

```
coverage exclude -add fsm.vhdl all
```

excludes all the transitions from the FSM's in that file.

```
coverage exclude -add fsm.vhdl -f fsm_test 0   2 3 4
```

excludes the transitions numbered 2, 3 and 4 from the FSM with Id 0 in the fsm_test design unit of the given file.

```
coverage exclude -add -inst /fsm_test/a1 -f 0 all
```

excludes all the transitions from the 0th FSM of the specified /fsm_test/a1 instance.

```
coverage exclude -add -inst /fsm_test/a1 -f 0 2 3 4
```

excludes all the transitions numbered 2, 3 and 4 from the 0th FSM of the specified instance.

Using -remove with coverage exclude

When the -remove option is used with coverage exclude, it re-enables the reporting of those transitions which have been excluded. The transitions are specified in the same manner as that for the coverage exclude command. For example:

```
coverage exclude -remove fsm.vhdl -f fsm_test 0    2 3 4
```

re-enables the reporting of the specified transitions.

Using the coverage clear command

The coverage clear command re-enables the reporting of those transitions which have been excluded by the user.

```
coverage clear -excluded -user
```

Chapter 16
C Debug

C Debug allows you to interactively debug FLI/PLI/VPI/DPI/SystemC/C/C++ source code with the open-source gdb debugger. Even though C Debug doesn't provide access to all gdb features, you may wish to read gdb documentation for additional information. For debugging memory errors in C source files, please refer to the application note entitled "Using the valgrind Tool with ModelSim".

Note

The functionality described in this chapter requires a cdebug license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Tip: Please be aware of the following caveats before using C Debug:

- C Debug is an interface to the open-source gdb debugger. We have not customized gdb source code, and C Debug doesn't remove any of the limitations or bugs of gdb.
- We assume that you are competent with C or C++ coding and C debugging in general.
- Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.
- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.
- Generally you should not have an existing *.gdbinit* file. If you do, make certain you haven't done any of the following: defined your own commands or renamed existing commands; used 'set annotate...', 'set height...', 'set width...', or 'set print...'; set breakpoints or watchpoints.
- To use C Debug on Windows platforms, you must compile your source code with gcc/g++. See Running C Debug on Windows Platforms below.

Supported Platforms and gdb Versions

ModelSim ships with the gdb 6.0 debugger. Testing has shown this version to be the most reliable for SystemC applications. However, for FLI/PLI/DPI applications, you can also use a current installation of gdb if you prefer.

For gcc versions 4.0 and above, gdb version 6.1 (or later) is required.

C Debug has been tested on these platforms with these versions of gdb:

Table 16-1. Supported Platforms and gdb Versions

Platform	**Required gdb version**
32-bit Solaris 8, 9[1]	gdb-5.0-sol-2.6
32- and 64-bit HP-UX 11.0[2], 11.11[3]	wdb version 3.3 or later
64-bit HP-UX B.11.22 on Itanium 2	wdb version 4.2
32-bit Redhat Linux 7.2 or later[1]	*/usr/bin/gdb* 5.2 or later
32-bit Windows 2000 and XP	gdb 6.0 from MinGW-32
Opteron / SuSE Linux 9.0 or Redhat EWS 3.0 (32-bit mode only)[1]	gdb 6.0 or later
x86 / Redhat Linux 6.0 to 7.1[1]	*/usr/bin/gdb* 5.2 or later
Opteron & Athlon 64 / Redhat EWS 3.0	gdb 5.3.92 or 6.1.1

[1] ModelSim ships gdb 6.3 for Solaris 8, 9 and Linux platforms.
[2] You must install kernel patch PHKL_22568 (or a later patch that supersedes PHKL_22568) on HP-UX 11.0. If you do not, you will see the following error message when trying to enable C Debug:
# Unable to find dynamic library list.
# error from C debugger
[3] You must install B.11.11.0306 Gold Base Patches for HP-UX 11i, June 2003.

To invoke C Debug, you must have the following:

- A *cdebug* license feature.
- The correct gdb debugger version for your platform.

Running C Debug on Windows Platforms

To use C Debug on Windows, you must compile your C/C++ source code using the gcc/g++ compiler supplied with ModelSim. Source compiled with Microsoft Visual C++ is not debuggable using C Debug.

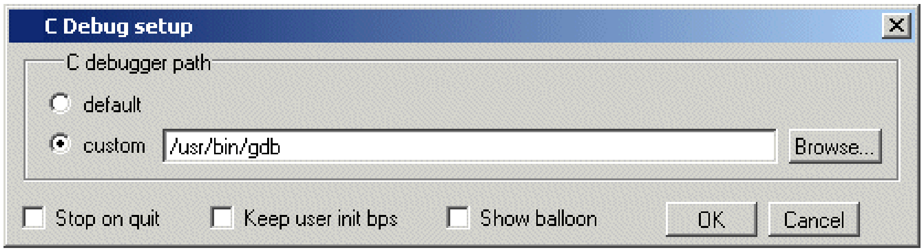
The g++ compiler is installed in the following location:

../modeltech/gcc-3.2.3-mingw32/

Setting Up C Debug

Before viewing your SystemC/C/C++ source code, you must set up the C Debug path and options. To set up C Debug, follow these steps:

1. Compile and link your C code with the **-g** switch (to create debug symbols) and without **-O** (or any other optimization switches you normally use). See SystemC Simulation for information on compiling and linking SystemC code. Refer to the chapter Verilog PLI/VPI/DPI for information on compiling and linking C code.
2. Specify the path to the gdb debugger by selecting **Tools** > **C Debug** > **C Debug Setup**.

   C Debug setup

   C debugger path

   ○ default

   ● custom /usr/bin/gdb

   □ Stop on quit □ Keep user init bps □ Show balloon

   Select "default" to point at the supplied version of gdb or "custom" to point at a separate installation.
3. Start the debugger by selecting **Tools** > **C Debug** > **Start C Debug**. ModelSim will start the debugger automatically if you set a breakpoint in a SystemC file.
4. If you are not using **gcc**, or otherwise haven't specified a source directory, specify a source directory for your C code with the following command:

   ModelSim> gdb dir <srcdirpath1>[:<srcdirpath2>[...]]

Running C Debug from a DO File

You can run C Debug from a DO file but there is a configuration issue of which you should be aware. It takes C Debug a few moments to start-up. If you try to execute a run command before C Debug is fully loaded, you may see an error like the following:

```
# ** Error: Stopped in C debugger, unable to real_run mti_run 10us
# Error in macro ./do_file line 8
# Stopped in C debugger, unable to real_run mti_run 10us
#     while executing
# "run 10us
```

In your DO file, add the command **cdbg_wait_for_starting** to alleviate this problem. For example:

```
cdbg enable_auto_step on
cdbg set_debugger /modelsim/5.8c_32/common/linux
cdbg debug_on
cdbg_wait_for_starting
run 10us
```

Setting Breakpoints

Breakpoints in C Debug work much like normal HDL breakpoints. You can create and edit them with ModelSim commands (bp, bd, enablebp, disablebp) or via a Source window in the GUI (see File-line breakpoints). Some differences do exist:

- The Breakpoints dialog in the ModelSim GUI doesn’t list C breakpoints.
- C breakpoint id numbers require a "c." prefix when referenced in a command.
- When using the bp command to set a breakpoint in a C file, you must use the **-c** argument.

Here are some example commands:

```
bp -c *0x400188d4
```

Sets a C breakpoint at the hex address 400188d4. Note the ’*’ prefix for the hex address.

```
bp -c or_checktf
```

Sets a C breakpoint at the entry to function **or_checktf**.

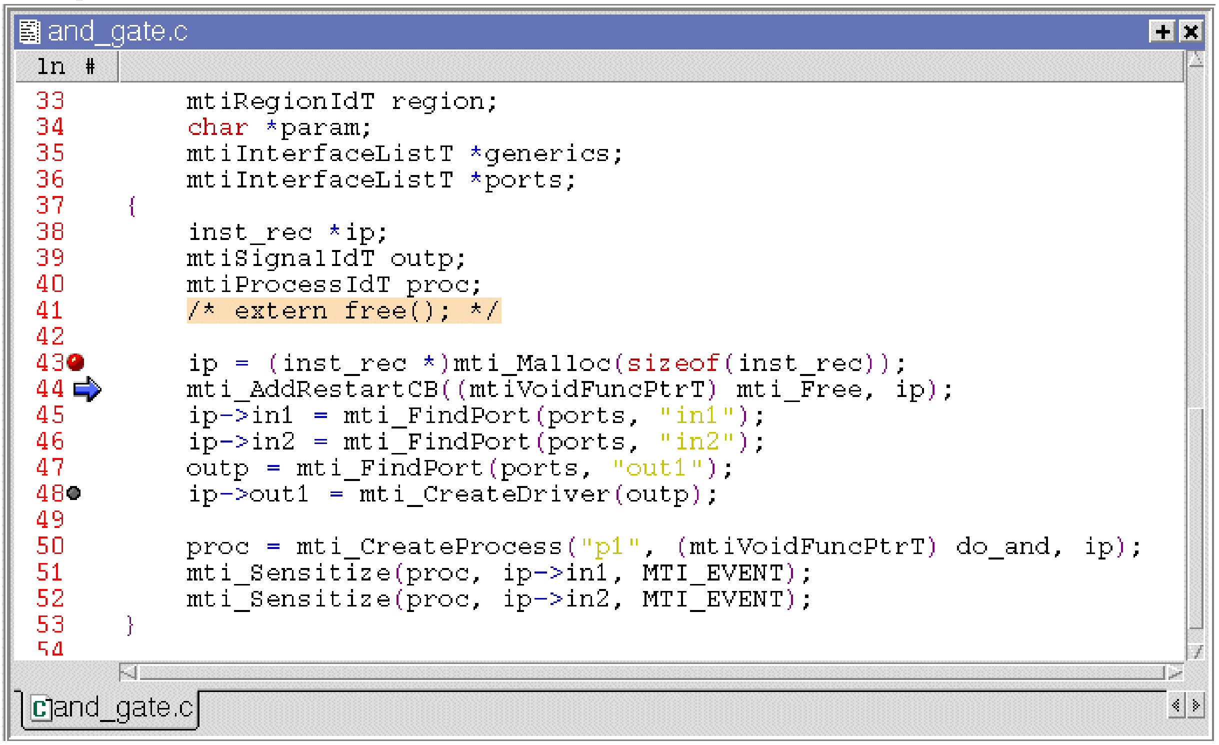
```
bp -c or.c 91
```

Sets a C breakpoint at line 91 of *or.c*.

```
enablebp c.1
```

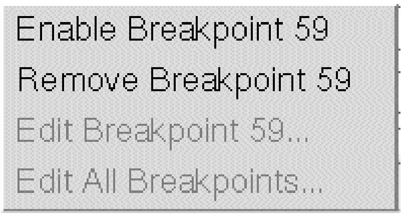
Enables C breakpoint number 1.

The graphic below shows a C file with one enabled breakpoint (on line 44) and one disabled breakpoint (on line 48).



```
33        mtiRegionIdT region;
34        char *param;
35        mtiInterfaceListT *generics;
36        mtiInterfaceListT *ports;
37    {
38        inst_rec *ip;
39        mtiSignalIdT outp;
40        mtiProcessIdT proc;
41        /* extern free(); */
42
43        ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44        mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45        ip->in1 = mti_FindPort(ports, "in1");
46        ip->in2 = mti_FindPort(ports, "in2");
47        outp = mti_FindPort(ports, "out1");
48        ip->out1 = mti_CreateDriver(outp);
49
50        proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
51        mti_Sensitize(proc, ip->in1, MTI_EVENT);
52        mti_Sensitize(proc, ip->in2, MTI_EVENT);
53    }
54
```

Clicking the red diamonds with your right (third) mouse button pops up a menu with commands for removing or enabling/disabling the breakpoints



Note

The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Do not set breakpoints in constructors of SystemC objects; it may crash the debugger.

Stepping in C Debug

Stepping in C Debug works much like you would expect. You use the same buttons and commands that you use when working with an HDL-only design.

Table 16-2.

Button	Menu equivalent	Other equivalents
Step steps the current simulation to the next statement; if the next statement is a call to a C function that was compiled with debug info, ModelSim will step into the function	Tools > C Debug > Run > Step	**use the step command at the CDBG> prompt** see: step command
Step Over statements are executed but treated as simple statements instead of entered and traced line-by-line; C functions are not stepped into unless you have an enabled breakpoint in the C file	Tools > C Debug > Run > Step -Over	**use the step -over command at the CDBG> prompt** see: step command
Continue Run continue the current simulation run until the end of the specified run length or until it hits a breakpoint or specified break event	Tools > C Debug > Run > Continue	**use the run -continue command at the CDBG> prompt** see: run command

Known Problems With Stepping in C Debug

The following are known limitations which relate to problems with gdb:

- The gdb debugger has a known bug that makes it impossible to set breakpoints reliably in constructors or destructors. Be careful while stepping through code which may end up calling constructors of SystemC objects; it may crash the debugger.

- With some platform and compiler versions, **step** may actually behave like **run -continue** when in a C file. This is a gdb quirk that results from not having any debugging information when in an internal function to VSIM (i.e., any FLI or VPI function). In these situations, use **step -over** to move line-by-line.

Quitting C Debug

To end a debugging session, you can do one of the following.

- From the GUI:

  Select **Tools** > **C Debug** > **Quit C Debug**.

- From the command line, enter the following in the Transcript window:

  ```
  cgdb quit
  ```

Note

Recommended usage is that you invoke C Debug once for a given simulation and then quit both C Debug and ModelSim. Starting and stopping C Debug more than once during a single simulation session may cause problems for gdb.

Finding Function Entry Points with Auto Find bp

ModelSim can automatically locate and set breakpoints at all currently known function entry points (i.e., PLI/VPI/DPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Select **Tools** > **C Debug** > **Auto find bp** to invoke this feature.

The **Auto find bp** command provides a "snapshot" of your design when you invoke the command. If additional callbacks get registered later in the simulation, ModelSim will not identify these new function entry points *unless* you re-execute the **Auto find bp** command. If you want functions to be identified regardless of when they are registered, use Identifying All Registered Function Calls instead.

The **Auto find bp** command sets breakpoints in an enabled state and doesn’t toggle that state to account for **step -over** or **run -continue** commands. This may result in unexpected behavior. For example, say you have invoked the **Auto find bp** command and you are currently stopped on a line of code that calls a C function. If you execute a **step -over** or **run -continue** command, ModelSim will stop on the breakpoint set in the called C file.

Identifying All Registered Function Calls

Auto step mode automatically identifies and sets breakpoints at registered function calls (i.e., PLI/VPI system tasks and functions and callbacks; and FLI subprograms and callbacks and processes created with **mti_CreateProcess**). Auto step mode is helpful when you are not entirely familiar with a design and its associated C routines. As you step through the design, ModelSim steps into and displays the associated C file when you hit a C function call in your HDL code. If you execute a **step -over** or **run -continue** command, ModelSim does not step into the C code.

When you first enable Auto step mode, ModelSim scans your design and sets enabled breakpoints at all currently known function entry points. As you step through the simulation, Auto step continues looking for newly registered callbacks and sets enabled breakpoints at any new entry points it identifies. Once you execute a **step -over** or **run -continue** command, Auto step disables the breakpoints it set, and the simulation continues running. The next time you execute a step command, the automatic breakpoints are re-enabled and Auto step sets breakpoints on any new entry points it identifies.

Note that Auto step does not disable user-set breakpoints.

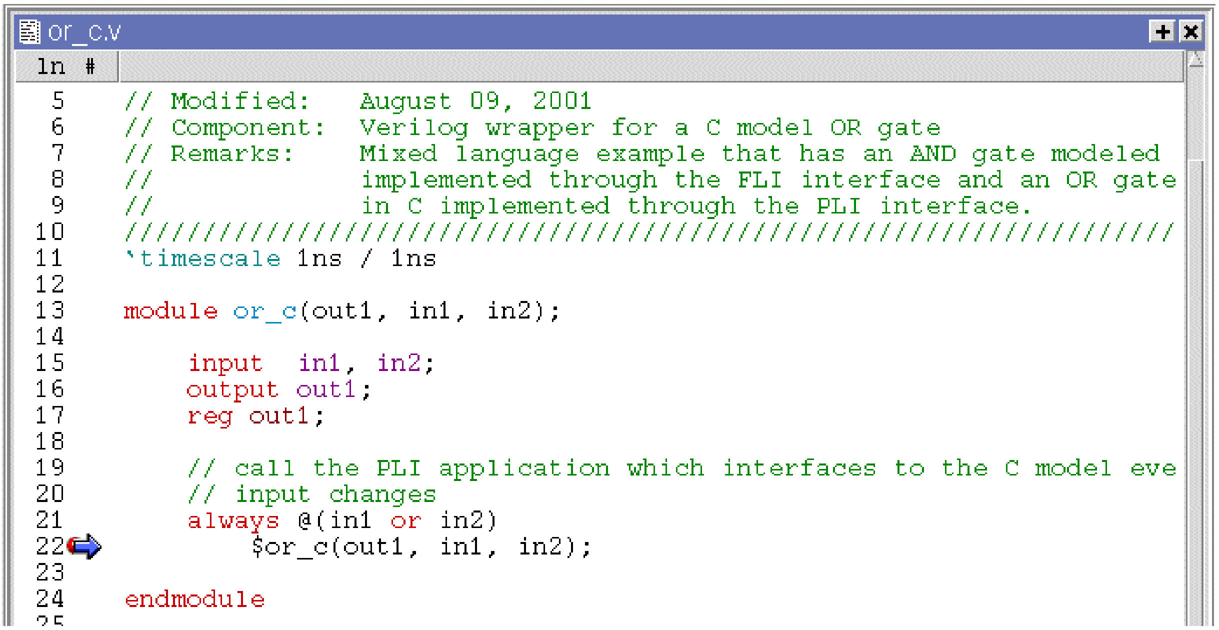
Enabling Auto Step Mode

To enable Auto step mode, follow these steps:

1. Configure C Debug as described in Setting Up C Debug.
2. Select **Tools** > **C Debug** > **Enable auto step**.
3. Load and run your design.

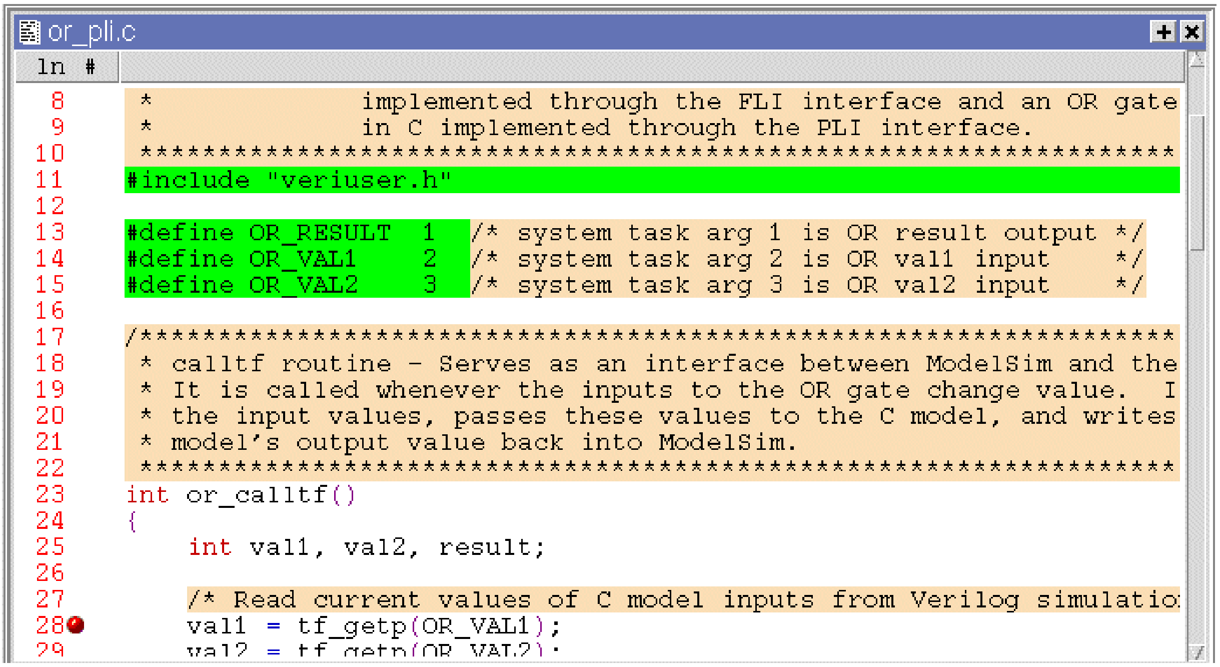
Example

The graphic below shows a simulation that has stopped at a user-set breakpoint on a PLI system task.



```
// Modified:   August 09, 2001
// Component:  Verilog wrapper for a C model OR gate
// Remarks:    Mixed language example that has an AND gate modeled
//             implemented through the FLI interface and an OR gate
//             in C implemented through the PLI interface.
////////////////////////////////////////////////////////////////////
`timescale 1ns / 1ns

module or_c(out1, in1, in2);

    input  in1, in2;
    output out1;
    reg out1;

    // call the PLI application which interfaces to the C model eve
    // input changes
    always @(in1 or in2)
        $or_c(out1, in1, in2);

endmodule
```

Because Auto step mode is enabled, ModelSim automatically sets a breakpoint in the underlying *xor_gate.c* file. If you click the step button at this point, ModelSim will step into that file.



```
 *              implemented through the FLI interface and an OR gate
 *              in C implemented through the PLI interface.
 ******************************************************************
#include "veriuser.h"

#define OR_RESULT   1   /* system task arg 1 is OR result output */
#define OR_VAL1     2   /* system task arg 2 is OR val1 input    */
#define OR_VAL2     3   /* system task arg 3 is OR val2 input    */

/******************************************************************
 * calltf routine - Serves as an interface between ModelSim and the
 * It is called whenever the inputs to the OR gate change value.  I
 * the input values, passes these values to the C model, and writes
 * model's output value back into ModelSim.
 ******************************************************************
int or_calltf()
{
    int val1, val2, result;

    /* Read current values of C model inputs from Verilog simulatio
    val1 = tf_getp(OR_VAL1);
    val2 = tf_getp(OR_VAL2);
```

Auto Find bp Versus Auto Step Mode

As noted in Finding Function Entry Points with Auto Find bp, the **Auto find bp** command also locates and sets breakpoints at function entry points. Note the following differences between Auto find bp and Auto step mode:

- Auto find bp provides a "snapshot" of currently known function entry points at the time you invoke the command. Auto step mode continues to locate and set automatic breakpoints in newly registered function calls as the simulation continues. In other words, Auto find bp is static while Auto step mode is dynamic.
- Auto find bp sets automatic breakpoints in an enabled state and doesn't change that state to account for step-over or run-continue commands. Auto step mode enables and disables automatic breakpoints depending on how you step through the design. In cases where you invoke both features, Auto step mode takes precedence over Auto find bp. In other words, even if Auto find bp has set enabled breakpoints, if you then invoke Auto step mode, it will toggle those breakpoints to account for step-over and run-continue commands.

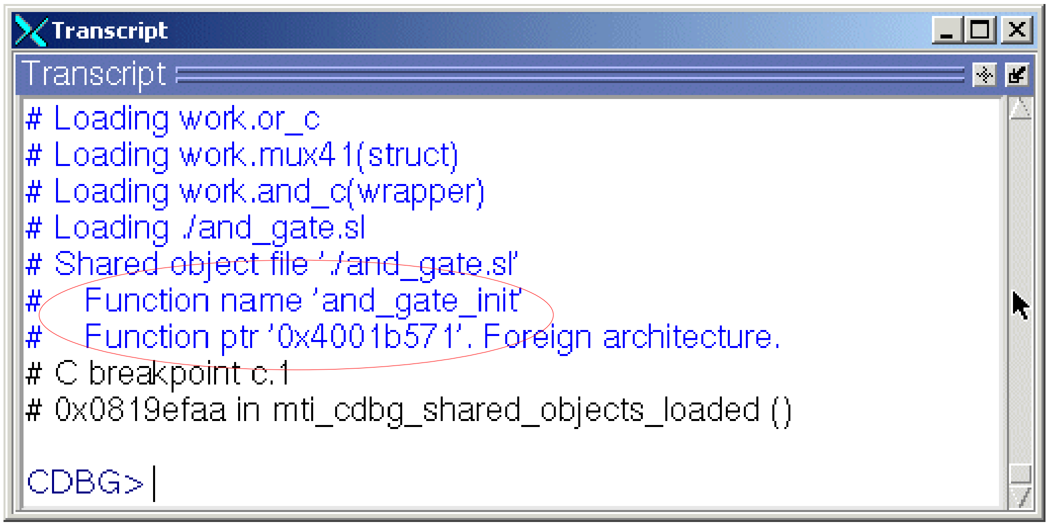
Debugging Functions During Elaboration

Initialization mode allows you to examine and debug functions that are called during elaboration (i.e., while your design is in the process of loading). When you select this mode, ModelSim sets special breakpoints for foreign architectures and PLI/VPI modules that allow you to set breakpoints in the initialization functions. When the design finishes loading, the special breakpoints are automatically deleted, and any breakpoints that you set are disabled (unless you specify **Keep user init bps** in the C debug setup dialog).

To run C Debug in initialization mode, follow these steps:

1. Start C Debug by selecting **Tools** > **C Debug** > **Start C Debug** *before* loading your design.
2. Select **Tools** > **C Debug** > **Init mode**.
3. Load your design.

As the design loads, ModelSim prints to the Transcript the names and/or hex addresses of called functions. For example the Transcript below shows a function pointer to a foreign architecture:



```
# Loading work.or_c
# Loading work.mux41(struct)
# Loading work.and_c(wrapper)
# Loading ./and_gate.sl
# Shared object file './and_gate.sl'
#     Function name 'and_gate_init'
#     Function ptr '0x4001b571'. Foreign architecture.
# C breakpoint c.1
# 0x0819efaa in mti_cdbg_shared_objects_loaded ()

CDBG>
```

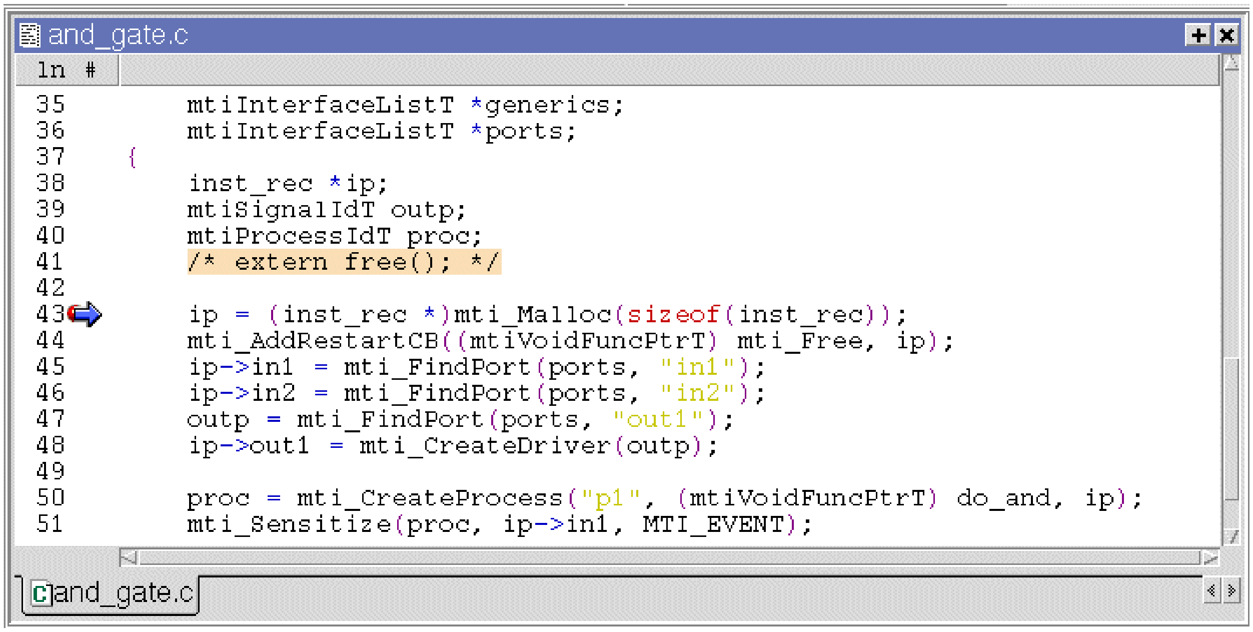
To set a breakpoint on that function, you would type:

```
bp -c *0x4001b571
```

or

```
bp -c and_gate_init
```

ModelSim in turn reports that it has set a breakpoint at line 37 of the *and_gate.c* file. As you continue through the design load using **run -continue**, ModelSim hits that breakpoint and displays the file and associated line in a Source window.



```
35        mtiInterfaceListT *generics;
36        mtiInterfaceListT *ports;
37    {
38        inst_rec *ip;
39        mtiSignalIdT outp;
40        mtiProcessIdT proc;
41        /* extern free(); */
42
43        ip = (inst_rec *)mti_Malloc(sizeof(inst_rec));
44        mti_AddRestartCB((mtiVoidFuncPtrT) mti_Free, ip);
45        ip->in1 = mti_FindPort(ports, "in1");
46        ip->in2 = mti_FindPort(ports, "in2");
47        outp = mti_FindPort(ports, "out1");
48        ip->out1 = mti_CreateDriver(outp);
49
50        proc = mti_CreateProcess("p1", (mtiVoidFuncPtrT) do_and, ip);
51        mti_Sensitize(proc, ip->in1, MTI_EVENT);
```

FLI Functions in Initialization Mode

There are two kinds of FLI functions that you may encounter in initialization mode. The first is a foreign architecture which was shown above. The second is a foreign function. ModelSim produces a Transcript message like the following when it encounters a foreign function during initialization:

```
# Shared object file './all.sl'
#    Function name 'in_params'
#    Function ptr '0x4001a950'. Foreign function.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., bp -c in_params) or the function pointer (i.e., bp -c *0x4001a950). Note, however, that foreign functions aren’t called during initialization. You would hit the breakpoint only during runtime and then only if you enabled the breakpoint after initialization was complete or had specified **Keep user init bps** in the C debug setup dialog.

PLI Functions in Initialization Mode

There are two methods for registering callback functions in the PLI: 1) using a veriusertfs array to define all usertf entries; and 2) adding an init_usertfs function to explicitly register each usertfs entry (see Registering PLI Applications for more details). The messages ModelSim produces in initialization mode vary depending on which method you use.

ModelSim produces a Transcript message like the following when it encounters a veriusertfs array during initialization:

```
# vsim -pli ./veriuser.sl mux_tb
# Loading ./veriuser.sl
# Shared object file './veriuser.sl'
#    veriusertfs array - registering calltf
#    Function ptr '0x40019518'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering checktf
#    Function ptr '0x40019570'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering sizetf
#    Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
cont
# Shared object file './veriuser.sl'
#    veriusertfs array - registering misctf
#    Function ptr '0x0'. $or_c.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set breakpoints on non-null callbacks using the function pointer (e.g., bp -c *0x40019570). You cannot set breakpoints on null functions. The sizetf and misctf entries in the example above are null (the function pointer is '0x0').

ModelSim reports the entries in multiples of four with at least one entry each for calltf, checktf, sizetf, and misctf. Checktf and sizetf functions are called during initialization but calltf and misctf are not called until runtime.

The second registration method uses init_usertfs functions for each usertfs entry. ModelSim produces a Transcript message like the following when it encounters an init_usertfs function during initialization:

```
# Shared object file './veriuser.sl'
#    Function name 'init_usertfs'
#    Function ptr '0x40019bec'. Before first call of init_usertfs.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using either the function name (i.e., bp -c init_usertfs) or the function pointer (i.e., bp -c *0x40019bec). ModelSim will hit this breakpoint as you continue through initialization.

VPI Functions in Initialization Mode

VPI functions are registered via routines placed in a table named vlog_startup_routines (see Registering PLI Applications for more details). ModelSim produces a Transcript message like the following when it encounters a vlog_startup_routines table during initialization:

```
# Shared object file './vpi_test.sl'
#    vlog_startup_routines array
#    Function ptr '0x4001d310'. Before first call using function pointer.
# C breakpoint c.1
# 0x0814fc96 in mti_cdbg_shared_objects_loaded ()
```

You can set a breakpoint on the function using the function pointer (i.e., bp -c *0x4001d310). ModelSim will hit this breakpoint as you continue through initialization.

Completing Design Load

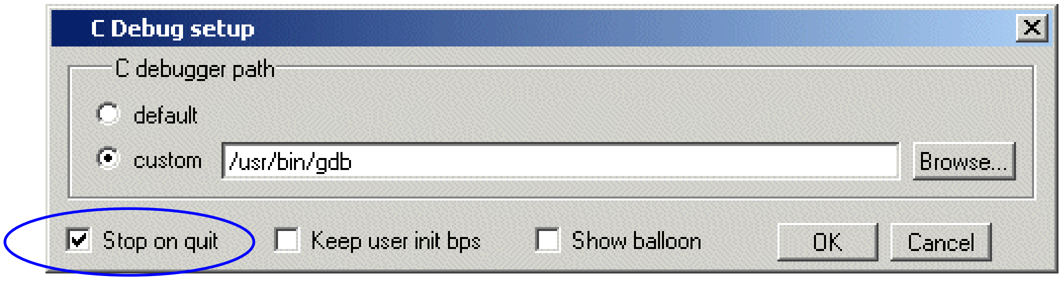
If you are through looking at the initialization code you can select **Tools** > **C Debug** > **Complete load** at any time, and ModelSim will continue loading the design without stopping. The one exception to this is if you have set a breakpoint in a LoadDone callback and also specified **Keep user init bps** in the C Debug setup dialog.

Debugging Functions when Quitting Simulation

Stop on quit mode allows you to debug functions that are called when the simulator exits. Such functions include those referenced by an **mti_AddQuitCB** function in FLI code, **misctf** function called by a quit or $finish in PLI code, or **cbEndofSimulation** function called by a quit or $finish in VPI code.

To enable Stop on quit mode, follow these steps:

1. Start C Debug by selecting **Tools** > **C Debug** > **Start C Debug**.
2. Select **Tools** > **C Debug** > **C Debug Setup**.
3. Select **Stop on quit** in the C Debug setup dialog.



C Debug setup

C debugger path

○ default

● custom /usr/bin/gdb

☑ Stop on quit □ Keep user init bps □ Show balloon

With this mode enabled, if you have set a breakpoint in a quit callback function, C Debug will stop at the breakpoint after you issue the quit command in ModelSim. This allows you to step and examine the code in the quit callback function.

Invoke **run -continue** when you are done looking at the C code.

Note that whether or not a C breakpoint was hit, when you return to the VSIM> prompt, you'll need to quit C Debug by selecting **Tools > C Debug > Quit C Debug** before finally quitting the simulation.

C Debug Command Reference

The table below provides a brief description of the commands that can be invoked when C Debug is running. Follow the links to the Reference Manual for complete command syntax.

Table 16-3.

Command	**Description**	**Corresponding menu command**
bd	deletes a previously set C breakpoint	right click breakpoint in Source window and select Remove Breakpoint
bp -c	sets a C breakpoint	click the desired line number in the Source window
change	changes the value of a C variable	none
describe	prints the type information of a C variable	select the C variable name in the Source window and select Tools > Describe or right click and select Describe.
disablebp	disables a previously set C breakpoint	right click breakpoint in Source window and select Disable Breakpoint
enablebp	enables a previously disabled C breakpoint	right click breakpoint in Source window and select Enable Breakpoint
examine	prints the value of a C variable	select the C variable name in the Source window and select Tools > Examine or right click and select Examine
gdb dir	sets the source directory search path for the C debugger	none
pop	moves the specified number of call frames up the C callstack	none

Table 16-3.

Command	Description	Corresponding menu command
push	moves the specified number of call frames down the C callstack	none
run -continue	continues running the simulation after stopping	click the run -continue button on the Main or Source window toolbar
run -finish	continues running the simulation until control returns to the calling function	Tools > C Debug > Run > Finish
show	displays the names and types of the local variables and arguments of the current C function	Tools > C Debug > Show
step	single step in the C debugger to the next executable line of C code; **step** goes into function calls, whereas **step -over** does not	click the step or step -over button on the Main or Source window toolbar
tb	displays a stack trace of the C call stack	Tools > C Debug > Traceback

Chapter 17
Profiling Performance and Memory Use

The ModelSim profiler combines a statistical sampling profiler with a memory allocation profiler to provide instance specific execution and memory allocation data. It allows you to quickly determine how your memory is being allocated and easily identify areas in your simulation where performance can be improved. The profiler can be used at all levels of design simulation – Functional, RTL, and Gate Level – and has the potential to save hours of regression test time. In addition, ASIC and FPGA design flows benefit from the use of this tool.

Note

The functionality described in this chapter requires a profiler license feature in your ModelSim license file. Please contact your Mentor Graphics sales representative if you currently do not have such a feature.

Profiling is not supported on Opteron / Athlon 64 platforms.

Introducing Performance and Memory Profiling

The profiler provides an interactive graphical representation of both memory and CPU usage on a per instance basis. It shows you what part of your design is consuming resources (CPU cycles or memory), allowing you to more quickly find problem areas in your code.

The profiler enables those familiar with the design and validation environment to find first-level improvements in a matter of minutes. For example, the statistical sampling profiler might show the following:

- non-accelerated VITAL library cells that are impacting simulation run time
- objects in the sensitivity list that are not required, resulting in a process that consumes more simulation time than necessary
- a testbench process that is active even though it is not needed
- an inefficient C module
- random number processes that are consuming simulation resources in a testbench running in non-random mode

With this information, you can make changes to the VHDL or Verilog source code that will speed up the simulation.

The memory allocation profiler provides insight into how much memory different parts of the design are consuming. The two major areas of concern are typically: 1) memory usage during elaboration, and 2) during simulation. If memory is exhausted during elaboration, for example, memory profiling may provide insights into what part(s) of the design are memory intensive. Or, if your HDL or PLI/FLI code is allocating memory and not freeing it when appropriate, the memory profiler will indicate excessive memory use in particular portions of the design.

Statistical Sampling Profiler

The profiler's statistical sampling profiler samples the current simulation at a user-determined rate (every <n> milliseconds of real or "wall-clock" time, not simulation time) and records what is executing at each sample point. The advantage of statistical sampling is that an entire simulation need not be run to get good information about what parts of your design are using the most simulation time. A few thousand samples, for example, can be accumulated before pausing the simulation to see where simulation time is being spent.

The statistical profiler reports only on the samples that it can attribute to user code. For example, if you use the **-nodebug** argument to vcom or vlog, it cannot report sample results.

Memory Allocation Profiler

The profiler's memory allocation profiler records every memory allocation and deallocation that takes place in the context of elaborating and simulating the design. It makes a record of the design element that is active at the time of allocation so memory resources can be attributed to appropriate parts of the design. This provides insights into memory usage that can help you re-code designs to, for example, minimize memory use, correct memory leaks, and change optimization parameters used at compile time.

Getting Started with the Profiler

Memory allocation profiling and statistical sampling are enabled separately.

Enabling the Memory Allocation Profiler

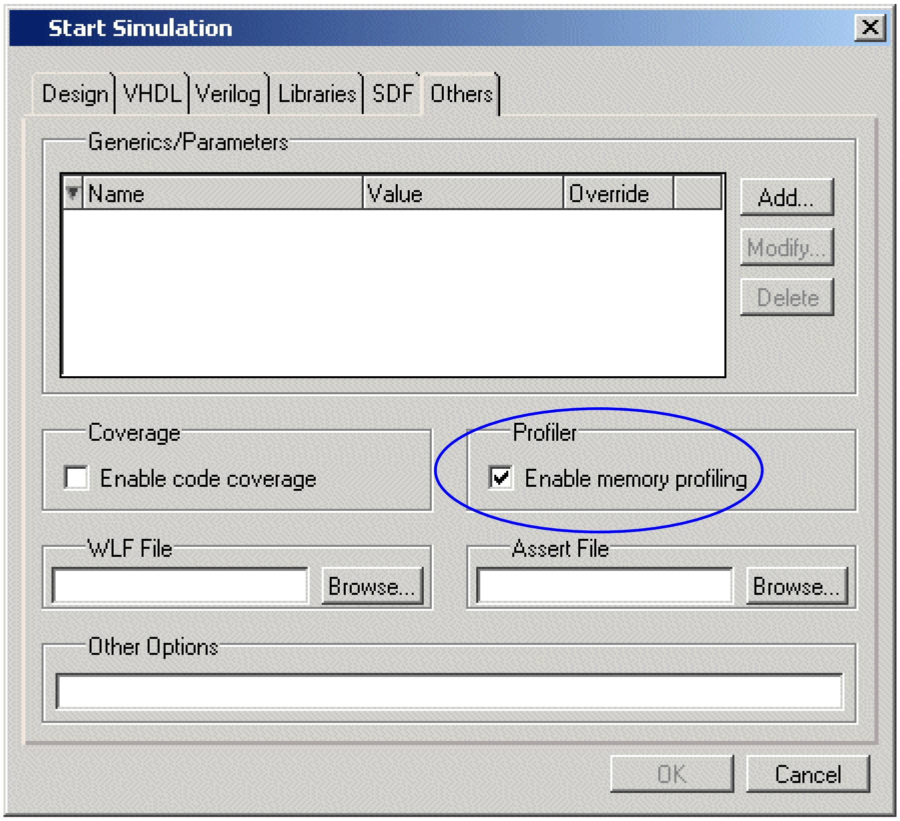
To record memory usage during elaboration and simulation, enable memory allocation profiling when the design is loaded with the **-memprof** argument to the **vsim** command.

```
vsim -memprof <design_unit>
```

Note that profile-data collection for the call tree is off by default. See The Call Tree View for additional information on collecting call-stack data.

You can use the graphic user interface as follows to perform the same task.

1. Select **Simulate > Start Simulation** or the Simulate icon, to open the Start Simulation dialog box.
2. Select the Others tab.
3. Click the **Enable memory profiling** checkbox to select it.



□ Enable code coverage

☑ Enable memory profiling

4. Click **OK** to load the design with memory allocation profiling enabled.

If memory allocation during elaboration is not a concern, the memory allocation profiler can be enabled at any time after the design is loaded by doing any one of the following:

- select **Tools** > **Profile** > **Memory**
- use the **-m** argument with the profile on command

```
profile on -m
```

- click the Memory Profiling icon

Handling Large Files

To allow memory allocation profiling of large designs, where the design itself plus the data required to keep track of memory allocation exceeds the memory available on the machine, the memory profiler allows you to route raw memory allocation data to an external file. This allows you to save the memory profile with minimal memory impact on the simulator, regardless of the size of your design.

The external data file is created during elaboration by using either the **-memprof+file=<filename>** or the **-memprof+fileonly=<filename>** argument with the vsim command .

The **-memprof+file=<filename>** option will collect memory profile data during both elaboration and simulation and save it to the named external file *and* makes the data available for viewing and reporting during the current simulation.

The **-memprof+fileonly=<filename>** option will collect memory profile data during both elaboration and simulation and save it to *only* the named external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

Alternatively, you can save memory profile data from the simulation only by using either the **-m -file <filename>** or the **-m -fileonly <filename>** argument with the profile on command.

The **-m -file <filename>** option saves memory profile data from simulation to the designated external file *and* makes the data available for viewing and reporting during the current simulation.

The **-m -fileonly <filename>** option saves memory profile data from simulation to *only* the designated external file. No data is saved for viewing and reporting during the current simulation, which reduces the overall amount of memory required by memory allocation profiling.

After elaboration and/or simulation is complete, a separate session can be invoked and the profile data can be read in with the profile reload command for analysis. It should be noted, however, that this command will clear all performance and memory profiling data collected to that point (implicit profile clear). Any currently loaded design will be unloaded (implicit quit **-sim**), and run-time profiling will be turned off (implicit profile off **-m -p**). If a new design is loaded after you have read the raw profile data, then all internal profile data is cleared (implicit profile clear), but run-time profiling is not turned back on.

Enabling the Statistical Sampling Profiler

To enable the profiler's statistical sampling profiler prior to a simulation run, do any one of the following:

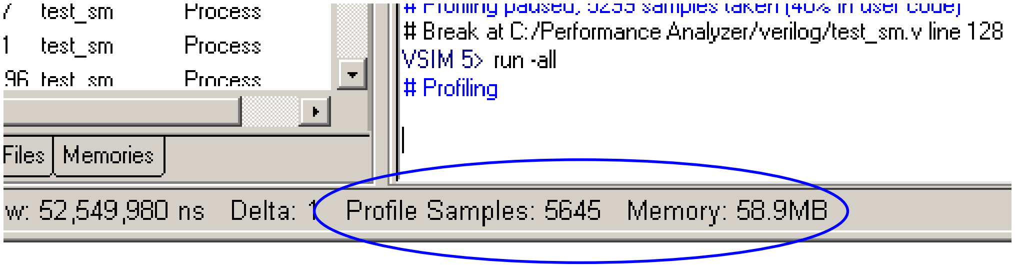
- select **Tools** > **Profile** > **Performance**
- use the profile on command
- click the Performance Profiling icon

Collecting Memory Allocation and Performance Data

Both memory allocation profiling and statistical sampling occur during the execution of a ModelSim **run** command. With profiling enabled, all subsequent **run** commands will collect memory allocation data and performance statistics. Profiling results are cumulative – each **run** command performed with profiling enabled will add new information to the data already gathered. To clear this data, select **Tools** > **Profile** > **Clear Profile Data** or use the profile clear command.

With the profiler enabled and a **run** command initiated, the simulator will provide a "Profiling" message in the transcript to indicate that profiling has started.

If the statistical sampling profiler and the memory allocation profiler are on, the status bar will display the number of Profile Samples collected and the amount of memory allocated, as shown below. Each profile sample will become a data point in the simulation's performance profile.



```
# Profiling paused, 5255 samples taken (40% in user code)
# Break at C:/Performance Analyzer/verilog/test_sm.v line 128
VSIM 5> run -all
# Profiling
```

Turning Profiling Off

You can turn off the statistical sampling profiler or the memory allocation profiler by doing any one of the following:

- deselect the **Performance** and/or **Memory** options in the **Tools** > **Profile menu**
- deselect the Performance Profiling and Memory Profiling icons in the toolbar

- use the profile off command with the **-p** or **-m** arguments.

Any ModelSim **run** commands that follow will not be profiled.

Running the Profiler on Windows with PLI/VPI Code

If you need to run the profiler under Windows on a design that contains FLI/PLI/VPI code, add these two switches to the compiling/linking command:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* file that the profiler can use in its report.

Interpreting Profiler Data

The utility of the data supplied by the profiler depends in large part on how your code is written. In cases where a single model or instance consumes a high percentage of simulation time or requires a high percentage of memory, the statistical sampling profiler or the memory allocation profiler quickly identifies that object, allowing you to implement a change that runs faster or requires less memory.

More commonly, simulation time or memory allocation will be spread among a handful of modules or entities – for example, 30% of simulation time split between models X, Y, and Z; or 20% of memory allocation going to models A, B, C and D. In such situations, careful examination and improvement of each model may result in overall speed improvement or more efficient memory allocation.

There are times, however, when the statistical sampling and memory allocation profilers tell you nothing more than that simulation time or memory allocation is fairly equally distributed throughout your design. In such situations, the profiler provides little helpful information and improvement must come from a higher level examination of how the design can be changed or optimized.

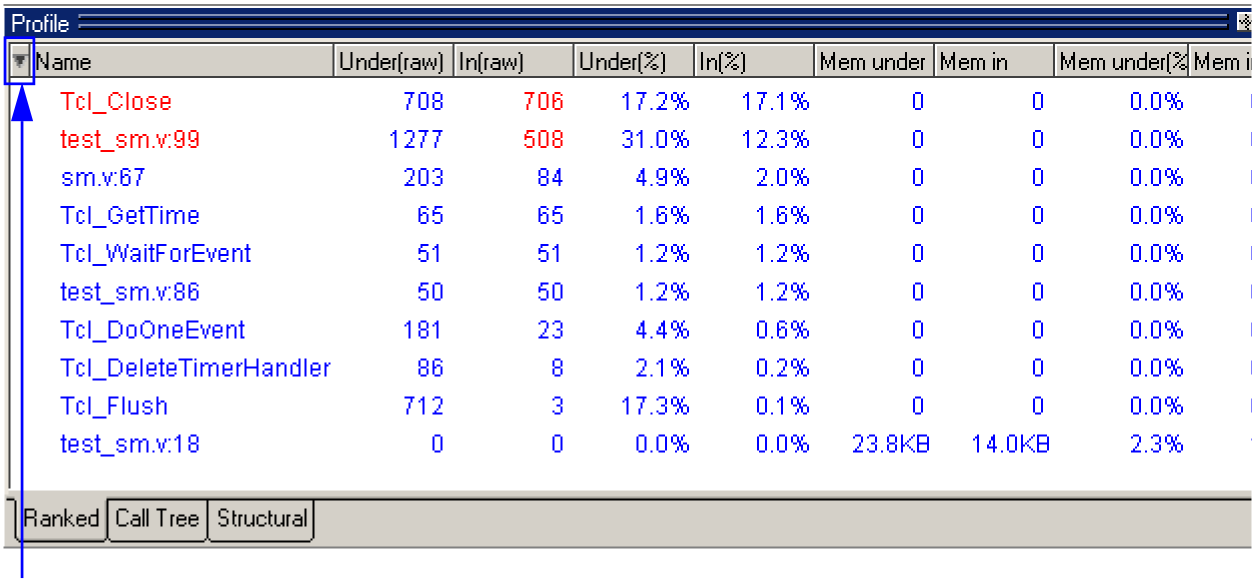
Viewing Profiler Results

The profiler provides three views of the collected data – *Ranked*, *Call Tree* and *Structural*. All three views are enabled by selecting **View > Windows > Profile** or by typing **view profilemain** at the VSIM prompt. This opens the Profile pane. The Profile pane includes selection tabs for the Ranked, Call Tree, and Structural views.

The Ranked View

The Ranked view displays the results of the statistical performance profiler and the memory allocation profiler for each function or instance. By default, ranked profiler results are sorted by values in the *In%* column, which shows the percentage of the total samples collected for each function or instance. You can sort ranked results by any other column by simply clicking the column heading. Click the down arrow to the left of the Name column to open a Configure Columns dialog, which allows you to select which columns are to be hidden or displayed.

The use of colors in the display provides an immediate visual indication of where your design is spending most of its simulation time. By default, red text indicates functions or instances that are consuming 5% or more of simulation time.



Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%	Mem i
Tcl_Close	708	706	17.2%	17.1%	0	0	0.0%	
test_sm.v:99	1277	508	31.0%	12.3%	0	0	0.0%	
sm.v:67	203	84	4.9%	2.0%	0	0	0.0%	
Tcl_GetTime	65	65	1.6%	1.6%	0	0	0.0%	
Tcl_WaitForEvent	51	51	1.2%	1.2%	0	0	0.0%	
test_sm.v:86	50	50	1.2%	1.2%	0	0	0.0%	
Tcl_DoOneEvent	181	23	4.4%	0.6%	0	0	0.0%	
Tcl_DeleteTimerHandler	86	8	2.1%	0.2%	0	0	0.0%	
Tcl_Flush	712	3	17.3%	0.1%	0	0	0.0%	
test_sm.v:18	0	0	0.0%	0.0%	23.8KB	14.0KB	2.3%	

Click here to hide or display columns

The Ranked view does not provide hierarchical, function-call information.

The Call Tree View

Data collection for the call tree is off by default, due to the fact that it will increase the simulation time and resource usage. Collection can be turned on from the VSIM command prompt with **profile option collect_calltrees on** and off with **profile option collect_calltrees off**. Call stack data collection can also be turned on with the **-memprof+call** argument to the vsim command.

By default, profiler results in the Call Tree view are sorted according to the *Under(%)* column, which shows the percentage of the total samples collected for each function or instance and all supporting routines or instances. Sort results by any other column by clicking the column heading. As in the Ranked view, red object names indicate functions or instances that, by default, are consuming 5% or more of simulation time.

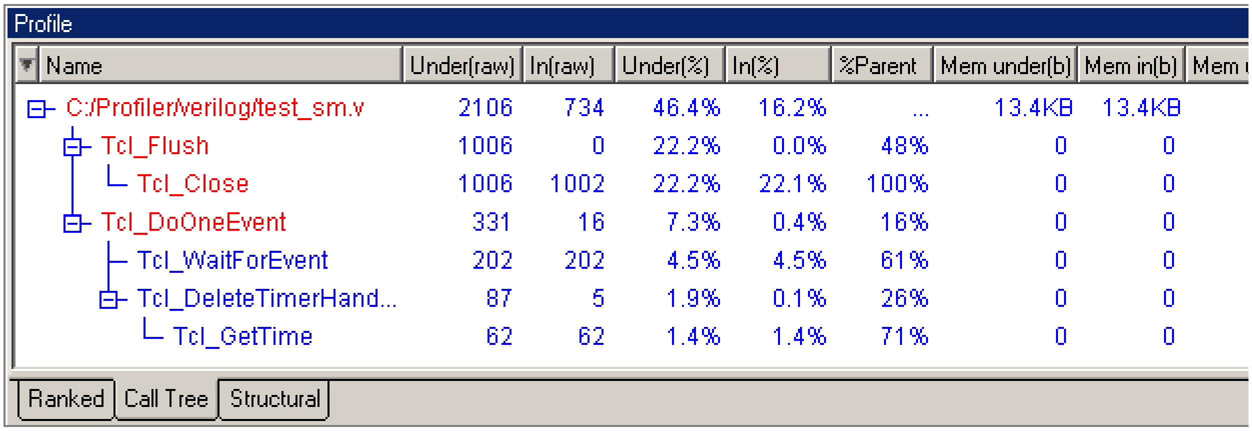
The Call Tree view differs from the Ranked view in two important respects.

- Entries in the Name column of the Call Tree view are indented in hierarchical order to indicate which functions or routines call which others.
- A *%Parent* column in the Call Tree view allows you to see what percentage of a parent routine's simulation time is used in which subroutines.

The Call Tree view presents data in a call-stack format that provides more context than does the ranked view about where simulation time is spent. For example, your models may contain several instances of a utility function that computes the maximum of 3-delay values. A Ranked view might reveal that the simulation spent 60% of its time in this utility function, but would not tell you which routine or routines were making the most use of it. The Call Tree view will reveal which line is calling the function most frequently. Using this information, you might decide that instead of calling the function every time to compute the maximum of the 3-delays, this spot in your VHDL code can be used to compute it just once. You can then store the maximum delay value in a local variable.

The two *%Parent* columns in the Call Tree view show the percent of simulation time or allocated memory a given function or instance is using of its parent's total simulation time or available memory. From these columns, you can calculate the percentage of total simulation time or memory taken up by any function. For example, if a particular parent entry used 10% of the total simulation time or allocated memory, and it called a routine that used 80% of its simulation time or memory, then the percentage of total simulation time spent in, or memory allocated to, that routine would be 80% of 10%, or 8%.

In addition to these differences, the Ranked view displays any particular function only once, regardless of where it was used. In the Call Tree view, the function can appear multiple times – each time in the context of where it was used.

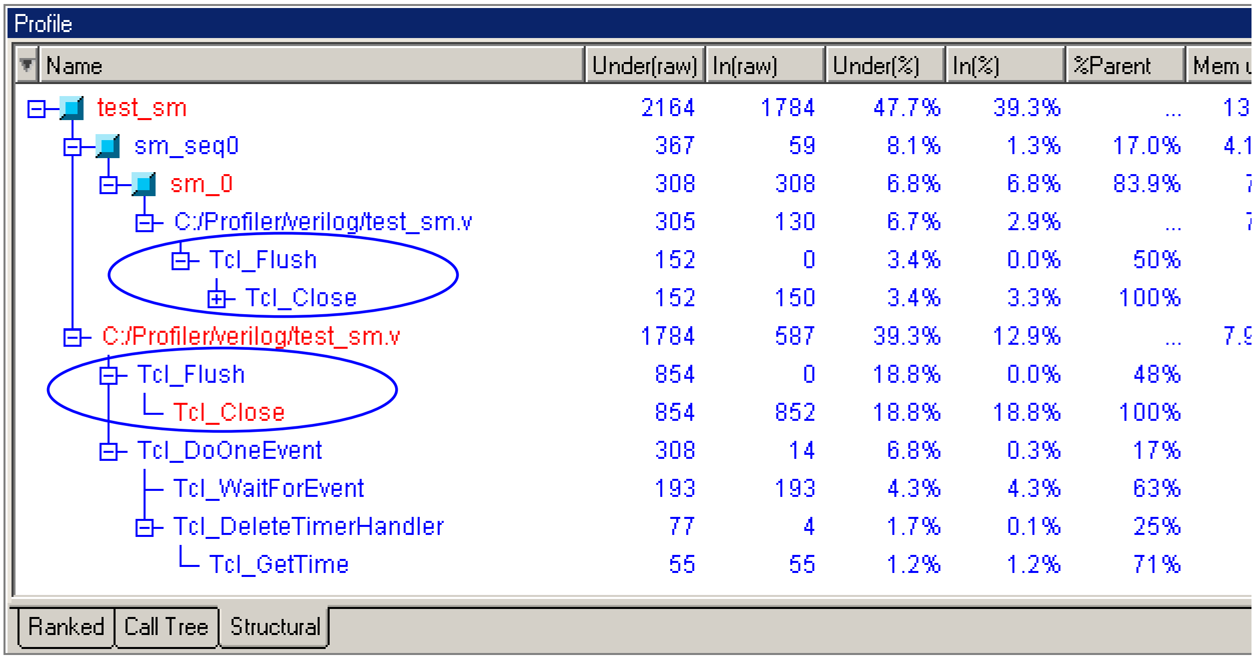


Profile

Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent	Mem under(b)	Mem in(b)	Mem
C:/Profiler/verilog/test_sm.v	2106	734	46.4%	16.2%	...	13.4KB	13.4KB	
Tcl_Flush	1006	0	22.2%	0.0%	48%	0	0	
Tcl_Close	1006	1002	22.2%	22.1%	100%	0	0	
Tcl_DoOneEvent	331	16	7.3%	0.4%	16%	0	0	
Tcl_WaitForEvent	202	202	4.5%	4.5%	61%	0	0	
Tcl_DeleteTimerHand...	87	5	1.9%	0.1%	26%	0	0	
Tcl_GetTime	62	62	1.4%	1.4%	71%	0	0	

The Structural View

The Structural view displays instance-specific performance and memory profile information in a hierarchical structure format identical to the structural view in the Workspace. It contains the same information found in the Call Tree view but adds an additional dimension with which to categorize performance samples and memory allocation. It shows how call stacks are associated with different instances in the design. For example, in the illustration that follows, *TCL_Flush* and *TCL_Close* appear under both *test_sm* and *sm_0*.



Name	Under(raw)	In(raw)	Under(%)	In(%)	%Parent	Mem u
test_sm	2164	1784	47.7%	39.3%	...	13
sm_seq0	367	59	8.1%	1.3%	17.0%	4.1
sm_0	308	308	6.8%	6.8%	83.9%	7
C:/Profiler/verilog/test_sm.v	305	130	6.7%	2.9%	...	7
Tcl_Flush	152	0	3.4%	0.0%	50%	
Tcl_Close	152	150	3.4%	3.3%	100%	
C:/Profiler/verilog/test_sm.v	1784	587	39.3%	12.9%	...	7.9
Tcl_Flush	854	0	18.8%	0.0%	48%	
Tcl_Close	854	852	18.8%	18.8%	100%	
Tcl_DoOneEvent	308	14	6.8%	0.3%	17%	
Tcl_WaitForEvent	193	193	4.3%	4.3%	63%	
Tcl_DeleteTimerHandler	77	4	1.7%	0.1%	25%	
Tcl_GetTime	55	55	1.2%	1.2%	71%	

In the Call Tree and Structural views, you can expand and collapse the various levels to hide data that is not useful to the current analysis and/or is cluttering the display. Click on the '+' box next to an object name to expand the hierarchy and show supporting functions and/or instances beneath it. Click the '-' box to collapse all levels beneath the entry.

Note that profile-data collection for the call tree is off by default. See The Call Tree View for additional information on collecting call-stack data.

You can also right click any function or instance in the Call Tree and Structural views to obtain popup menu selections for rooting the display to the currently selected item, to ascend the displayed root one level, or to expand and collapse the hierarchy.

Toggling Display of Call Stack Entries

By default call stack entries do not display in the Structural tab. To display call stack entries, right-click in the pane and select **Show Calls**.

Viewing Profile Details

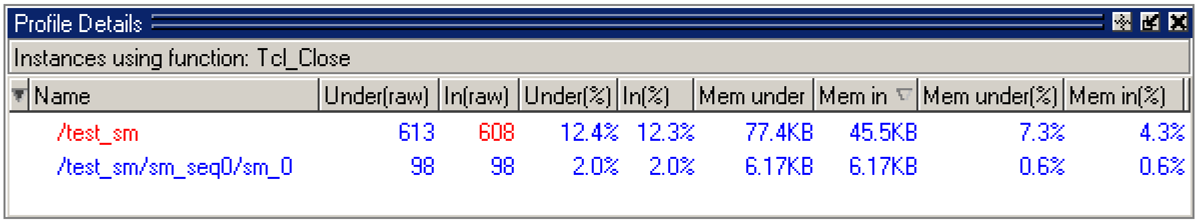
The Profiler increases visibility into simulation performance and memory usage with dynamic links to the Source window and the Profile Details pane. The Profile Details pane is enabled by selecting **View** > **Windows > Profile Details** or by entering the **view profiledetails** command at the VSIM prompt. You can also right-click any function or instance in the Ranked, Call Tree, or Structural views to open a popup menu that includes options for viewing profile details. The following options are available:

View Source

When View Source is selected the Source window opens to the location of the selected function in the source code.

Function Usage

When Function Usage is selected, the Profile Details pane opens and displays all instances using the selected function. In the Profile Details pane shown below, all the instances using function *Tcl_Close* are displayed. The statistical performance and memory allocation data shows how much simulation time and memory is used by *Tcl_Close* in each instance.



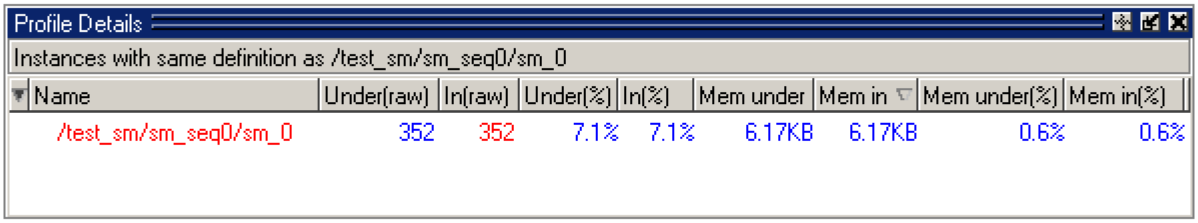
Profile Details

Instances using function: Tcl_Close

Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
/test_sm	613	608	12.4%	12.3%	77.4KB	45.5KB	7.3%	4.3%
/test_sm/sm_seq0/sm_0	98	98	2.0%	2.0%	6.17KB	6.17KB	0.6%	0.6%

Instance Usage

When Instance Usage is selected all instances with the same definition as the selected instance will be displayed in the Profile Details pane.



Profile Details

Instances with same definition as /test_sm/sm_seq0/sm_0

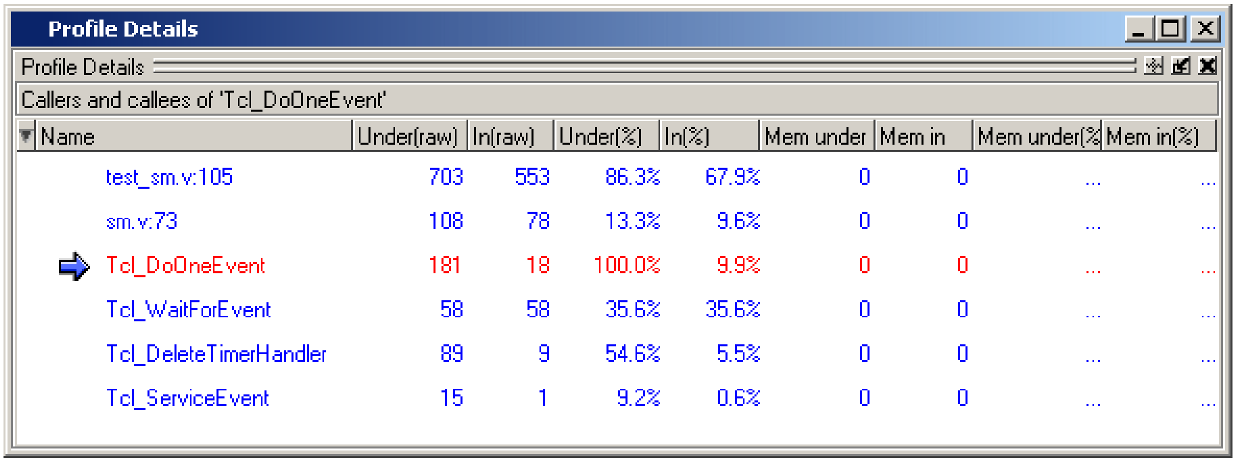
Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%)	Mem in(%)
/test_sm/sm_seq0/sm_0	352	352	7.1%	7.1%	6.17KB	6.17KB	0.6%	0.6%

View Instantiation

When View Instantiation is selected the Source window opens to the point in the source code where the selected instance is instantiated.

Callers and Callees

When Callers & Callees is selected, callers and callees for the selected function are displayed in the Profile Details window. Items above the selected function are callers; items below are callees. The selected function is distinguished with an arrow on the left and in 'hotForeground' color as shown below.



Callers and callees of 'Tcl_DoOneEvent'

Name	Under(raw)	In(raw)	Under(%)	In(%)	Mem under	Mem in	Mem under(%	Mem in(%)
test_sm.v:105	703	553	86.3%	67.9%	0	0	...	...
sm.v:73	108	78	13.3%	9.6%	0	0	...	...
➡ Tcl_DoOneEvent	181	18	100.0%	9.9%	0	0	...	...
Tcl_WaitForEvent	58	58	35.6%	35.6%	0	0	...	...
Tcl_DeleteTimerHandler	89	9	54.6%	5.5%	0	0	...	...
Tcl_ServiceEvent	15	1	9.2%	0.6%	0	0	...	...

Display in Call Tree

When Display in Call Tree is selected the Call Tree view of the Profile window expands to display all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

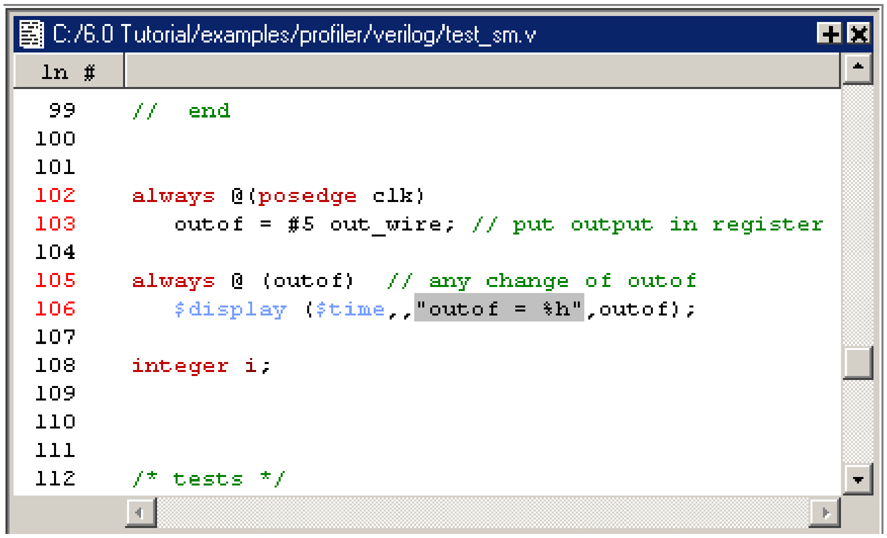
Note that profile-data collection for the call tree is off by default. See The Call Tree View for additional information on collecting call-stack data.

Display in Structural

When Display in Structural is selected the Structural view of the Profile window expands to display all occurrences of the selected function and puts the selected function into a search buffer so you can easily cycle across all occurrences of that function.

Integration with Source Windows

The Ranked, Call Tree and Structural profile views are all dynamically linked to Source window. You can double-click any function or instance in the Ranked, Call Tree and Structural views to bring up that object in a Source window with the selected line highlighted.



```
//  end


always @(posedge clk)
   outof = #5 out_wire; // put output in register

always @ (outof)   // any change of outof
   $display ($time,,"outof = %h",outof);

integer i;



/* tests */
```

You can perform the same task by right-clicking any function or instance in any one of the three Profile views and selecting View Source from the popup menu that opens.

When you right-click an instance in the Structural profile view, the View Instantiation selection will become active in the popup menu. Selecting this option opens the instantiation in a Source window and highlights it.

The right-click popup menu also allows you to change the root instance of the display, ascend to the next highest root instance, or reset the root instance to the top level instance.

The selection of a context in the structure tab of the Workspace pane will cause the root display to be set in the Structural view.

Analyzing C Code Performance

You can include C code in your design via SystemC, the Verilog PLI/VPI, or the ModelSim FLI. The profiler can be used to determine the impact of these C modules on simulator performance. Factors that can affect simulator performance when a design includes C code are as follows:

- PLI/FLI applications with large sensitivity lists
- Calling operating system functions from C code
- Calling the simulator's command interpreter from C code

- Inefficient C code

In addition, the Verilog PLI/VPI requires maintenance of the simulator's internal data structures as well as the PLI/VPI data structures for portability. (VHDL does not have this problem in ModelSim because the FLI gets information directly from the simulator.)

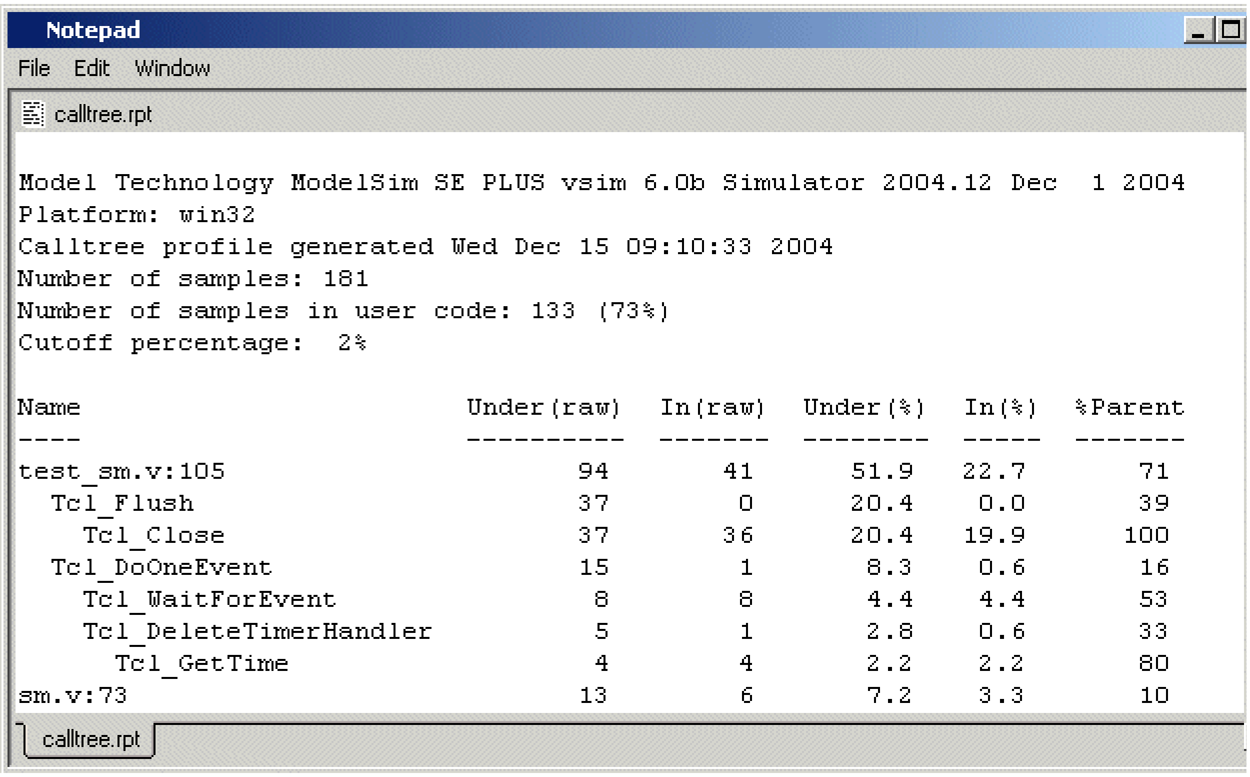
Reporting Profiler Results

You can create performance and memory profile reports using the Profile Report dialog or the profile report command.

For example, the command

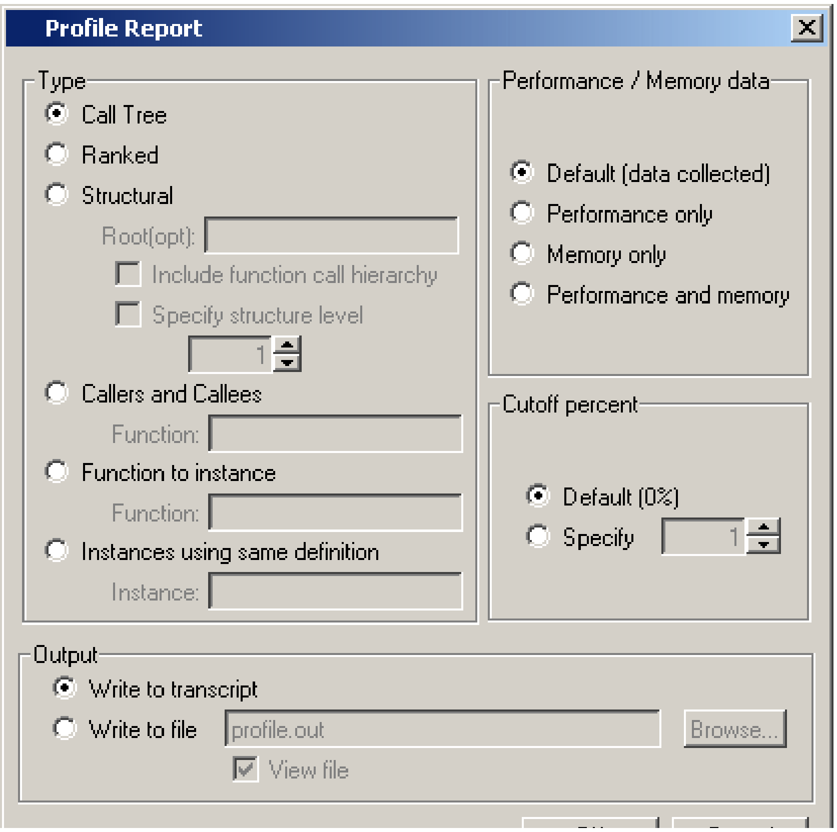
profile report -calltree -file calltree.rpt -cutoff 2

will produce a Call Tree profile report in a text file called *calltree.rpt*, as shown here.



```
Model Technology ModelSim SE PLUS vsim 6.0b Simulator 2004.12 Dec  1 2004
Platform: win32
Calltree profile generated Wed Dec 15 09:10:33 2004
Number of samples: 181
Number of samples in user code: 133 (73%)
Cutoff percentage:  2%

Name                        Under(raw)  In(raw)  Under(%)  In(%)  %Parent
----                        ----------  -------  --------  -----  -------
test_sm.v:105                       94       41      51.9   22.7       71
  Tcl_Flush                         37        0      20.4    0.0       39
    Tcl_Close                       37       36      20.4   19.9      100
  Tcl_DoOneEvent                    15        1       8.3    0.6       16
    Tcl_WaitForEvent                 8        8       4.4    4.4       53
    Tcl_DeleteTimerHandler           5        1       2.8    0.6       33
      Tcl_GetTime                    4        4       2.2    2.2       80
sm.v:73                             13        6       7.2    3.3       10
```

Select **Tools** > **Profile** > **Profile Report** to open the Profile Report dialog. From the dialog below, a Structural profile report will be created from the root instance pathname, */test_sm/sm_seq0*. The report will include function call hierarchy and three structure levels. Both performance and memory data will be displayed with a cutoff of 3% - meaning, the report will not contain any functions or instances that do not use 3% or more of simulation time or memory. The report will be written to a file called *profile.out* and, since the "View file" box is selected, it will be generated and displayed in Notepad when the OK button is clicked.



Profile Report

Type
- ● Call Tree
- ○ Ranked
- ○ Structural
  - Root(opt):
  - ☐ Include function call hierarchy
  - ☐ Specify structure level: 1
- ○ Callers and Callees
  - Function:
- ○ Function to instance
  - Function:
- ○ Instances using same definition
  - Instance:

Performance / Memory data
- ● Default (data collected)
- ○ Performance only
- ○ Memory only
- ○ Performance and memory

Cutoff percent
- ● Default (0%)
- ○ Specify: 1

Output
- ● Write to transcript
- ○ Write to file: profile.out
  - ☑ View file

Chapter 18
Signal Spy

The Verilog language allows access to any signal from any other hierarchical block without having to route it via the interface. This means you can use hierarchical notation to either assign or determine the value of a signal in the design hierarchy from a testbench. This capability fails when a Verilog testbench attempts to reference a signal in a VHDL block or reference a signal in a Verilog block through a VHDL level of hierarchy.

This limitation exists because VHDL does not allow hierarchical notation. In order to reference internal hierarchical signals, you have to resort to defining signals in a global package and then utilize those signals in the hierarchical blocks in question. But, this requires that you keep making changes depending on the signals that you want to reference.

The Signal Spy procedures and system tasks overcome the aforementioned limitations. They allow you to monitor (spy), drive, force, or release hierarchical objects in a VHDL or mixed design.

The VHDL procedures are provided via the Util Package within the *modelsim_lib* library. To access the procedures you would add lines like the following to your VHDL code:

```
library modelsim_lib;
use modelsim_lib.util.all;
```

The Verilog tasks are available as built-in System Tasks and Functions. The table below shows the VHDL procedures and their corresponding Verilog system tasks.

Table 18-1.

VHDL procedures	Verilog system tasks
disable_signal_spy	$disable_signal_spy
enable_signal_spy	$enable_signal_spy
init_signal_driver	$init_signal_driver
init_signal_spy	$init_signal_spy
signal_force	$signal_force
signal_release	$signal_release

Designed for Testbenches

Signal Spy limits the portability of your code. HDL code with Signal Spy procedures or tasks works only in ModelSim, not other simulators. We therefore recommend using Signal Spy only in testbenches, where portability is less of a concern, and the need for such a tool is more applicable.

disable_signal_spy

The disable_signal_spy() procedure disables the associated init_signal_spy. The association between the disable_signal_spy call and the init_signal_spy call is based on specifying the same src_object and dest_object arguments to both functions. The disable_signal_spy call can only affect init_signal_spy calls that had their control_state argument set to "0" or "1".

Syntax

disable_signal_spy(src_object, dest_object, verbose)

Returns

Nothing

Arguments

Table 18-2.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred. Default is 0, no message

Related procedures

init_signal_spy, enable_signal_spy

Example

See init_signal_spy init_signal_spy Example

enable_signal_spy

The enable_signal_spy() procedure enables the associated init_signal_spy. The association between the enable_signal_spy call and the init_signal_spy call is based on specifying the same src_object and dest_object arguments to both functions. The enable_signal_spy call can only affect init_signal_spy calls that had their control_state argument set to "0" or "1".

Syntax

enable_signal_spy(src_object, dest_object, verbose)

Returns

Nothing

Arguments

Table 18-3.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred. Default is 0, no message

Related procedures

init_signal_spy, disable_signal_spy

Example

See init_signal_spy init_signal_spy Example

init_signal_driver

The init_signal_driver() procedure drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_driver procedure drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the init_signal_driver value in the resolution of the signal.

Call only once

The init_signal_driver procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_driver only once for a particular pair of signals. Once init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all init_signal_driver calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_driver calls and a simple wait statement. The process will execute once and then wait forever. See the example below.

Syntax

init_signal_driver(src_object, dest_object, delay, delay_type, verbose)

Returns

Nothing

Arguments

Table 18-4.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
delay	time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	del_mode	Optional. Specifies the type of delay that will be applied. The value must be either mti_inertial or mti_transport. The default is mti_inertial.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related procedures

init_signal_spy, signal_force, signal_release

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to *mti_transport*, the setting will be ignored and the delay type will be *mti_inertial*.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.

init_signal_driver Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
  signal clk0 : std_logic;

begin

  gen_clk0 : process
  begin
    clk0 <= '1' after 0 ps, '0' after 20 ps;
    wait for 40 ps;
  end process gen_clk0;

  drive_sig_process : process
  begin
    init_signal_driver("clk0", "/testbench/uut/blk1/clk", open, open, 1);
    init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100 ps,
                       mti_transport);
    wait;
  end process drive_sig_process;

  ...

end;
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *open* entries allow the default delay and delay_type while setting the verbose parameter to a 1. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps.

init_signal_spy

The init_signal_spy() procedure mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a VHDL architecture (e.g., a testbench).

The init_signal_spy procedure only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value that was set by init_signal_spy.

Call only once

The init_signal_spy procedure creates a persistent relationship between the source and destination signals. Hence, you need to call init_signal_spy once for a particular pair of signals. Once init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the control_state is set.

The control_state determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the enable_signal_spy and disable_signal_spy calls.

We recommend that you place all init_signal_spy calls in a VHDL process. You need to code the VHDL process correctly so that it is executed only once. The VHDL process should not be sensitive to any signals and should contain only init_signal_spy calls and a simple wait statement. The process will execute once and then wait forever, which is the desired behavior. See the example below.

Syntax

init_signal_spy(src_object, dest_object, verbose, control_state)

Returns

Nothing

Arguments

Table 18-5.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.
control_state	integer	Optional. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state. The default is -1, no ability to enable/disable and mirroring is enabled. "0" turns on the ability to enable/disable and initially disables mirroring. "1" turns on the ability to enable/disable and initially enables mirroring.

Related procedures

init_signal_driver, signal_force, signal_release, enable_signal_spy, disable_signal_spy

Limitations

- When mirroring the value of a Verilog register/net onto a VHDL signal, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

init_signal_spy Example

```
library ieee;
library modelsim_lib;
use ieee.std_logic_1164.all;
use modelsim_lib.util.all;
entity top is
end;
architecture only of top is
  signal top_sig1 : std_logic;
```

```
begin
  ...
  spy_process : process
  begin
    init_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",1,1);
    wait;
  end process spy_process;
  ...
  spy_enable_disable : process(enable_sig)
  begin
    if (enable_sig = '1') then
      enable_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",0);
    elseif (enable_sig = '0')
      disable_signal_spy("/top/uut/inst1/sig1","/top/top_sig1",0);
    end if;
  end process spy_enable_disable;
  ...
end;
```

In this example, the value of */top/uut/inst1/sig1* is mirrored onto */top/top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the init_signal_spy is initially enabled.

The mirroring of values will be disabled when enable_sig transitions to a ’0’ and enable when enable_sig transitions to a ’1’.

signal_force

The signal_force() procedure forces the value specified onto an existing VHDL signal or Verilog register or net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_force works the same as the force command with the exception that you cannot issue a repeating force. The force will remain on the signal until a signal_release, a force or release command, or a subsequent signal_force is issued. Signal_force can be called concurrently or sequentially in a process.

This command acquires displays any signals using your radix setting (either the default, or as you specify) unless you specify the radix in the *value* you set.

Syntax

signal_force(dest_object, value, rel_time, force_type, cancel_period, verbose)

Returns

Nothing

Arguments

Table 18-6.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	forcetype	Optional. Specifies the type of force that will be applied. The value must be one of the following; default, deposit, drive, or freeze. The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command for further details on force type.
cancel_period	time	Optional. Cancels the signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1 ms. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related procedures

init_signal_driver, init_signal_spy, signal_release

Limitations

You cannot force bits or slices of a register; you can force only the entire register.

signal_force Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is
begin

  force_process : process
  begin
    signal_force("/testbench/uut/blk1/reset", "1", 0 ns, freeze, open, 1);
    signal_force("/testbench/uut/blk1/reset", "0", 40 ns, freeze, 2 ms,
1);
    wait;
  end process force_process;

  ...

end;
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 2 ms after the second signal_force call was executed.

If you want to skip parameters so that you can specify subsequent parameters, you need to use the keyword "open" as a placeholder for the skipped parameter(s). The first signal_force procedure illustrates this, where an "open" for the cancel_period parameter means that the default value of -1 ms is used.

signal_release

The signal_release() procedure releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers or nets at any level of the design hierarchy from within a VHDL architecture (e.g., a testbench).

A signal_release works the same as the noforce command. Signal_release can be called concurrently or sequentially in a process.

Syntax

signal_release(dest_object, verbose)

Returns

Nothing

Arguments

Table 18-7.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related procedures

init_signal_driver, init_signal_spy, signal_force

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

signal_release Example

```
library IEEE, modelsim_lib;
use IEEE.std_logic_1164.all;
use modelsim_lib.util.all;

entity testbench is
end;

architecture only of testbench is

  signal release_flag : std_logic;

begin

  stim_design : process
  begin
    ...
    wait until release_flag = '1';
    signal_release("/testbench/dut/blk1/data", 1);
    signal_release("/testbench/dut/blk1/clk", 1);
    ...
  end process stim_design;

  ...

end;
```

The above example releases any forces on the signals data and *clk* when the signal *release_flag* is a "1". Both calls will send a message to the transcript stating which signal was released and when.

$disable_signal_spy

The $disable_signal_spy() system task disables the associated $init_signal_spy task. The association between the $disable_signal_spy task and the $init_signal_spy task is based on specifying the same src_object and dest_object arguments to both tasks. The $disable_signal_spy task can only affect $init_signal_spy tasks that had their control_state argument set to "0" or "1".

Syntax

$disable_signal_spy(src_object, dest_object, verbose)

Returns

Nothing

Arguments

Table 18-8.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to disable.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that a disable occurred and the simulation time that it occurred. Default is 0, no message

Related tasks

$init_signal_spy, $enable_signal_spy

Example

See $init_signal_spy $init_signal_spy Example

$enable_signal_spy

The $enable_signal_spy() system task enables the associated $init_signal_spy task. The association between the $enable_signal_spy task and the $init_signal_spy task is based on specifying the same src_object and dest_object arguments to both tasks. The $enable_signal_spy task can only affect $init_signal_spys tasks that had their control_state argument set to "0" or "1".

Syntax

$enable_signal_spy(src_object, dest_object, verbose)

Returns

Nothing

Arguments

Table 18-9.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. This path should match the path that was specified in the init_signal_spy call that you wish to enable.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the transcript stating that an enable occurred and the simulation time that it occurred. Default is 0, no message

Related tasks

$init_signal_spy, $disable_signal_spy

Example

See $init_signal_spy $init_signal_spy Example

$init_signal_driver

The $init_signal_driver() system task drives the value of a VHDL signal or Verilog net (called the src_object) onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to drive signals or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_driver system task drives the value onto the destination signal just as if the signals were directly connected in the HDL code. Any existing or subsequent drive or force of the destination signal, by some other means, will be considered with the $init_signal_driver value in the resolution of the signal.

Call only once

The $init_signal_driver system task creates a persistent relationship between the source and destination signals. Hence, you need to call $init_signal_driver only once for a particular pair of signals. Once $init_signal_driver is called, any change on the source signal will be driven on the destination signal until the end of the simulation.

Thus, we recommend that you place all $init_signal_driver calls in a Verilog initial block. See the example below.

Syntax

$init_signal_driver(src_object, dest_object, delay, delay_type, verbose)

Returns

Nothing

Arguments

Table 18-10.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
delay	integer, real, or time	Optional. Specifies a delay relative to the time at which the src_object changes. The delay can be an inertial or transport delay. If no delay is specified, then a delay of zero is assumed.
delay_type	integer	Optional. Specifies the type of delay that will be applied. The value must be either 0 (inertial) or 1 (transport). The default is 0.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object is driving the dest_object. Default is 0, no message.

Related tasks

$init_signal_spy, $signal_force, $signal_release

Limitations

- When driving a Verilog net, the only *delay_type* allowed is inertial. If you set the delay type to 1 (transport), the setting will be ignored, and the delay type will be inertial.
- Any delays that are set to a value less than the simulator resolution will be rounded to the nearest resolution unit; no special warning will be issued.
- Verilog memories (arrays of registers) are not supported.

$init_signal_driver Example

```
`timescale 1 ps / 1 ps

module testbench;

reg clk0;

initial begin
  clk0 = 1;
  forever begin
   #20 clk0 = ~clk0;
  end
end

initial begin
   $init_signal_driver("clk0", "/testbench/uut/blk1/clk", , , 1);
   $init_signal_driver("clk0", "/testbench/uut/blk2/clk", 100, 1);
end

  ...

endmodule
```

The above example creates a local clock (*clk0*) and connects it to two clocks within the design hierarchy. The *.../blk1/clk* will match local *clk0* and a message will be displayed. The *.../blk2/clk* will match the local *clk0* but be delayed by 100 ps. For the second call to work, the *.../blk2/clk* must be a VHDL based signal, because if it were a Verilog net a 100 ps inertial delay would consume the 40 ps clock period. Verilog nets are limited to only inertial delays and thus the setting of 1 (transport delay) would be ignored.

$init_signal_spy

The $init_signal_spy() system task mirrors the value of a VHDL signal or Verilog register/net (called the src_object) onto an existing VHDL signal or Verilog register (called the dest_object). This allows you to reference signals, registers, or nets at any level of hierarchy from within a Verilog module (e.g., a testbench).

The $init_signal_spy system task only sets the value onto the destination signal and does not drive or force the value. Any existing or subsequent drive or force of the destination signal, by some other means, will override the value set by $init_signal_spy.

Call only once

The $init_signal_spy system task creates a persistent relationship between the source and the destination signal. Hence, you need to call $init_signal_spy only once for a particular pair of signals. Once $init_signal_spy is called, any change on the source signal will mirror on the destination signal until the end of the simulation unless the control_state is set.

The control_state determines whether the mirroring of values can be enabled/disabled and what the initial state is. Subsequent control of whether the mirroring of values is enabled/disabled is handled by the $enable_signal_spy and $disable_signal_spy tasks.

We recommend that you place all $init_signal_spy tasks in a Verilog initial block. See the example below.

Syntax

$init_signal_spy(src_object, dest_object, verbose, control_state)

Returns

Nothing

Arguments

Table 18-11.

Name	Type	Description
src_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to a Verilog register or VHDL signal. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the src_object's value is mirrored onto the dest_object. Default is 0, no message.
control_state	integer	Optional. Possible values are -1, 0, or 1. Specifies whether or not you want the ability to enable/disable mirroring of values and, if so, specifies the initial state. The default is -1, no ability to enable/disable and mirroring is enabled. "0" turns on the ability to enable/disable and initially disables mirroring. "1" turns on the ability to enable/disable and initially enables mirroring.

Related tasks

$init_signal_driver, $signal_force, $signal_release, $disable_signal_spy

Limitations

- When mirroring the value of a VHDL signal onto a Verilog register, the VHDL signal must be of type bit, bit_vector, std_logic, or std_logic_vector.
- Verilog memories (arrays of registers) are not supported.

$init_signal_spy Example

```
module top;
...
reg top_sig1;
reg enable_reg;
...
initial
  begin
  $init_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",1,1);
  end
```

```
    always @ (posedge enable_reg)
    begin
    $enable_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",0);
    end
    always @ (negedge enable_reg)
    begin
    $disable_signal_spy(".top.uut.inst1.sig1",".top.top_sig1",0);
    end
  ...
  endmodule
```

In this example, the value of *.top.uut.inst1.sig1* is mirrored onto *.top.top_sig1*. A message is issued to the transcript. The ability to control the mirroring of values is turned on and the init_signal_spy is initially enabled.

The mirroring of values will be disabled when enable_reg transitions to a ’0’ and enabled when enable_reg transitions to a ’1’.

$signal_force

The $signal_force() system task forces the value specified onto an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to force signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_force works the same as the force command with the exception that you cannot issue a repeating force. The force will remain on the signal until a $signal_release, a force or release command, or a subsequent $signal_force is issued. $signal_force can be called concurrently or sequentially in a process.

Syntax

$signal_force(dest_object, value, rel_time, force_type, cancel_period, verbose)

Returns

Nothing

Arguments

Table 18-12.

Name	Type	Description
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
value	string	Required. Specifies the value to which the dest_object is to be forced. The specified value must be appropriate for the type.
rel_time	integer, real, or time	Optional. Specifies a time relative to the current simulation time for the force to occur. The default is 0.
force_type	integer	Optional. Specifies the type of force that will be applied. The value must be one of the following; 0 (default), 1 (deposit), 2 (drive), or 3 (freeze). The default is "default" (which is "freeze" for unresolved objects or "drive" for resolved objects). See the force command for further details on force type.
cancel_period	integer, real, time	Optional. Cancels the $signal_force command after the specified period of time units. Cancellation occurs at the last simulation delta cycle of a time unit. A value of zero cancels the force at the end of the current time period. Default is -1. A negative value means that the force will not be cancelled.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the value is being forced on the dest_object at the specified time. Default is 0, no message.

Related tasks

$init_signal_driver, $init_signal_spy, $signal_release

Limitations

- You cannot force bits or slices of a register; you can force only the entire register.
- Verilog memories (arrays of registers) are not supported.

$signal_force Example

```
`timescale 1 ns / 1 ns

module testbench;

initial
  begin
   $signal_force("/testbench/uut/blk1/reset", "1", 0, 3, , 1);
   $signal_force("/testbench/uut/blk1/reset", "0", 40, 3, 200000, 1);
  end

...

endmodule
```

The above example forces *reset* to a "1" from time 0 ns to 40 ns. At 40 ns, *reset* is forced to a "0", 200000 ns after the second $signal_force call was executed.

$signal_release

The $signal_release() system task releases any force that was applied to an existing VHDL signal or Verilog register/net (called the dest_object). This allows you to release signals, registers, or nets at any level of the design hierarchy from within a Verilog module (e.g., a testbench).

A $signal_release works the same as the noforce command. $signal_release can be called concurrently or sequentially in a process.

Syntax

$signal_release(dest_object, verbose)

Returns

Nothing

Arguments

Table 18-13.

Name	**Type**	**Description**
dest_object	string	Required. A full hierarchical path (or relative downward path with reference to the calling block) to an existing VHDL signal or Verilog register/net. Use the path separator to which your simulation is set (i.e., "/" or "."). A full hierarchical path must begin with a "/" or ".". The path must be contained within double quotes.
verbose	integer	Optional. Possible values are 0 or 1. Specifies whether you want a message reported in the Transcript stating that the signal is being released and the time of the release. Default is 0, no message.

Related tasks

$init_signal_driver, $init_signal_spy, $signal_force

Limitations

- You cannot release a bit or slice of a register; you can release only the entire register.

$signal_release Example

```
module testbench;

reg release_flag;

always @(posedge release_flag) begin
  $signal_release("/testbench/dut/blk1/data", 1);
  $signal_release("/testbench/dut/blk1/clk", 1);
end

...

endmodule
```

The above example releases any forces on the signals *data* and *clk* when the register *release_flag* transitions to a "1". Both calls will send a message to the transcript stating which signal was released and when.

Chapter 19
Monitoring Simulations with JobSpy

This chapter describes JobSpy™, a tool for monitoring and controlling batch simulations and simulation farms.

Designers frequently run multiple simulation jobs in batch mode once verification reaches the regression testing stage. They face the problem that simulation farms and batch-mode runs offer little visibility into and control over simulation jobs. JobSpy helps alleviate this problem by allowing you to interact with batch jobs. By creating a process external to the running simulator, JobSpy can send and receive information about the running jobs.

Some applications of JobSpy include the following:

- Checking a simulation to see how far along it is.
- Examining internal signal values to check if the design is functioning correctly, without stopping the simulation.
- Suspending one job to release a license for a more important job. You can restart the suspended job later.
- Instructing the running batch job to do a checkpoint of the job and then carry on running. If the workstation that was running a batch job were to fail at sometime in the future, you would be possible to restart the job again from the saved checkpoint file.

You can run JobSpy from the command line, from within the ModelSim GUI, or from a standalone GUI. The actual commands that are sent and received across the communication pipe are the same for all modes of operation. The GUI simply provides a dialog box where you can see all the running jobs.

Who Can Control Jobs?

Any person who knows what port@host to set their JOBSPY_DAEMON variable to can control jobs submitted to that host. The intended use is that a person would set their JOBSPY_DAEMON variable, start the daemon, and then only they could control their jobs (unless they told somebody what port@host to use). Each user can use his/her own port id to monitor only their jobs.

Basic JobSpy Flow

The basic steps for setting up and using JobSpy are as follows:

1. Set JOBSPY_DAEMON environment variable.
2. Start JobSpy daemon.

   Command line: **jobspy -startd**

   GUI: **Tools** > **JobSpy** > **Daemon** > **Start**
3. Start simulation jobs.
4. Use **jobspy** command or Job Manager GUI to monitor results.

Starting the JobSpy Daemon

The JobSpy daemon must be started prior to launching any simulation jobs. The daemon tracks jobs by setting up a communication pipe with each running simulation. When a simulation job starts, the daemon opens a TCP/IP port for the job and then records in a file the port number, the host name that the job was started on, and the working directory. With a connection to the job established, you can invoke various commands via the command line or GUI to monitor or control the job.

There are two steps to starting the daemon:

1. Set the JOBSPY_DAEMON environment variable.

   The environment variable is set with the following syntax:

   ```
   JOBSPY_DAEMON=<port_NUMBER>@<host>
   ```

   For example,

   ```
   JOBSPY_DAEMON=1301@rhino
   ```

   Every user who will run JobSpy must set this environment variable. You will typically set this in a start-up script (e.g., *.cshrc* file) so that every new shell will have access to the daemon.
2. Invoke the daemon using the **jobspy -startd** command or by selecting **Tools** > **JobSpy** > **Daemon** > **Start Daemon** from within ModelSim.

Running JobSpy from the Command Line

The JobSpy command-line interface is accessible from a shell prompt or within the ModelSim GUI. There is only one command to remember--**jobspy**.

Command Syntax

jobspy [-gui] [-killd] [-startd] | jobs | status | <jobid> <command>

See the jobspy command for complete syntax. A few points you of which you should be aware:

- **jobspy -startd** invokes the daemon
- **jobspy jobs** lists all jobs and their id numbers; you need the ids in order to execute commands on the jobs
- <jobid> <command> allows you to issue commands to a job; only certain commands can be used, as noted below

Simulation Commands Available to JobSpy

A select number of simulator commands can be run on jobs via JobSpy. The table below lists the available commands with a brief description.

Table 19-1.

Command	**Description**
stop	stops a simulation
go	resumes a stopped job
checkpoint or check	checkpoints a simulation
savewlf	saves simulation results to a WLF file; see Viewing Results During Active Simulation; by default this command uses the pathname from the remote machine
examine	prints the value of a signal in the remote job
log	logs signals in the waveform log file (.wlf)
nolog	removes logged signals from the waveform log file (.wlf)
now	prints job's current simulation time
pwd	prints the job's current working directory
quit	exits a simulation (terminates job)
set	sets a TCL variable in the remote job's interpreter
simstatus	shows simulation's current status
suspend	suspends job (releases license)
unsuspend	un-suspends job (reacquires license)

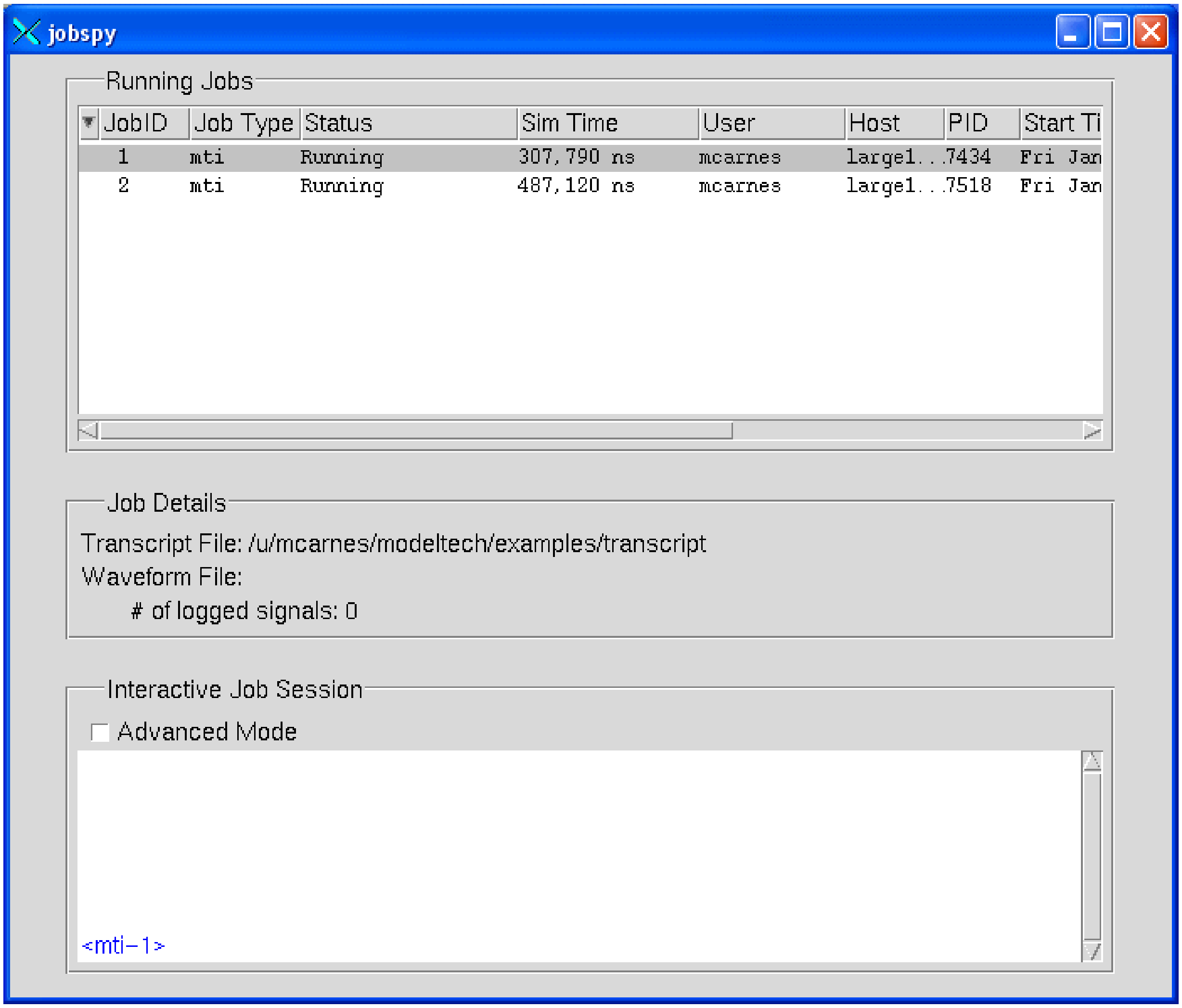
Example Session

The following example illustrates a mock session of JobSpy:

```
$ JOBSPY_DAEMON=1300@time //sets the daemon to a port@host
$ export JOBSPY_DAEMON //exports the environment variable

$ jobspy -startd //start the daemon

$ jobspy jobs // print list of jobs
JobID  Type  Sim Status  Sim Time  user  Host  PID   Start Time
   Directory
5     mti   Running    1,200ns   alla time  24710 Mon Dec 27.. /u/alla/z
11    mti   Running    3,433ns   mcar larg  24915 Tue Dec 28.. /u/mcar/x

$ jobspy 11 checkpoint //checkpoint job 11
Checkpointing Job

$ jobspy 11 cont //resume job 11
continuing

$ jobspy 5 dataset save sim snap.wlf // saving waveforms from job 5
Dataset "sim" exported as WLF file: snap.wlf. @ 1,200ns

$ vsim -view snap.wlf // viewing waveforms from job 5
```

Running the JobSpy GUI

JobSpy includes a GUI called Job Manager that can be invoked within ModelSim or separately as a stand-alone tool. The Job Manager shows all active simulations in real time and provides convenient access to JobSpy commands. The picture below shows Job Manager.



JobID	Job Type	Status	Sim Time	User	Host	PID	Start Ti
1	mti	Running	307,790 ns	mcarnes	large1. .	.7434	Fri Jan
2	mti	Running	487,120 ns	mcarnes	large1. .	.7518	Fri Jan

Transcript File: /u/mcarnes/modeltech/examples/transcript
Waveform File:
# of logged signals: 0

□ Advanced Mode

```
<mti-1>
```

Starting Job Manager

You can start Job Manager from a shell prompt or from within ModelSim:

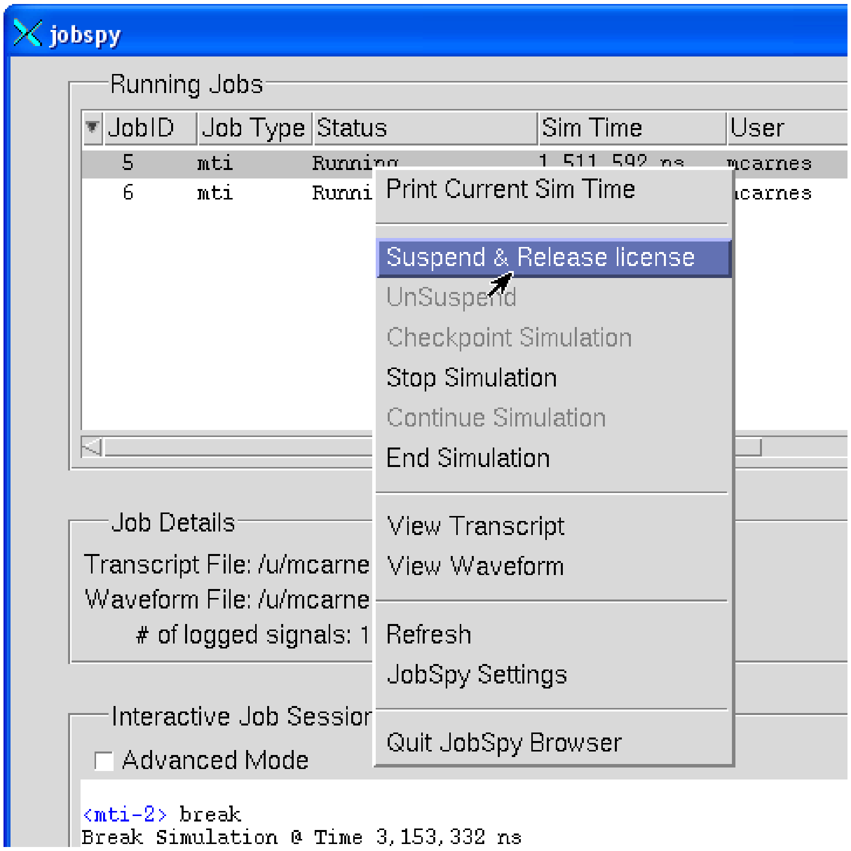
Table 19-2.

Location	Command
Shell prompt	**jobspy -gui**
ModelSim GUI	**Tools** > **JobSpy** > **JobSpy Job Manager**

Invoking Commands in Job Manager

You can invoke commands in Job Manager via a shortcut menu or from the prompt in the Interactive Job Session window.

Right-click (3rd mouse button) a job in the list to access menu commands:



JobID	Job Type	Status	Sim Time	User
5	mti	Running	1,511,592 ns	mcarnes
6	mti	Runni		carnes

Transcript File: /u/mcarne
Waveform File: /u/mcarne
# of logged signals: 1

□ Advanced Mode

```
<mti-2> break
Break Simulation @ Time 3,153,332 ns
```

Interactive Job Session Pane

The Interactive Job Session pane provides a command line for interacting with jobs. Commands you enter affect the job currently selected in the top of the dialog. For example, in the graphic above, job 5 is selected. If I want to suspend the job, I can type "suspend" at the <mti-#> prompt to suspend job 5 and release the simulation license.

See Simulation Commands Available to JobSpy for a list of commands you can enter in the Interactive Job Session pane.

Note

If you check Advanced Mode, you can enter any ModelSim command at the prompt. However, you need to be careful as many ModelSim commands will not function properly with JobSpy.

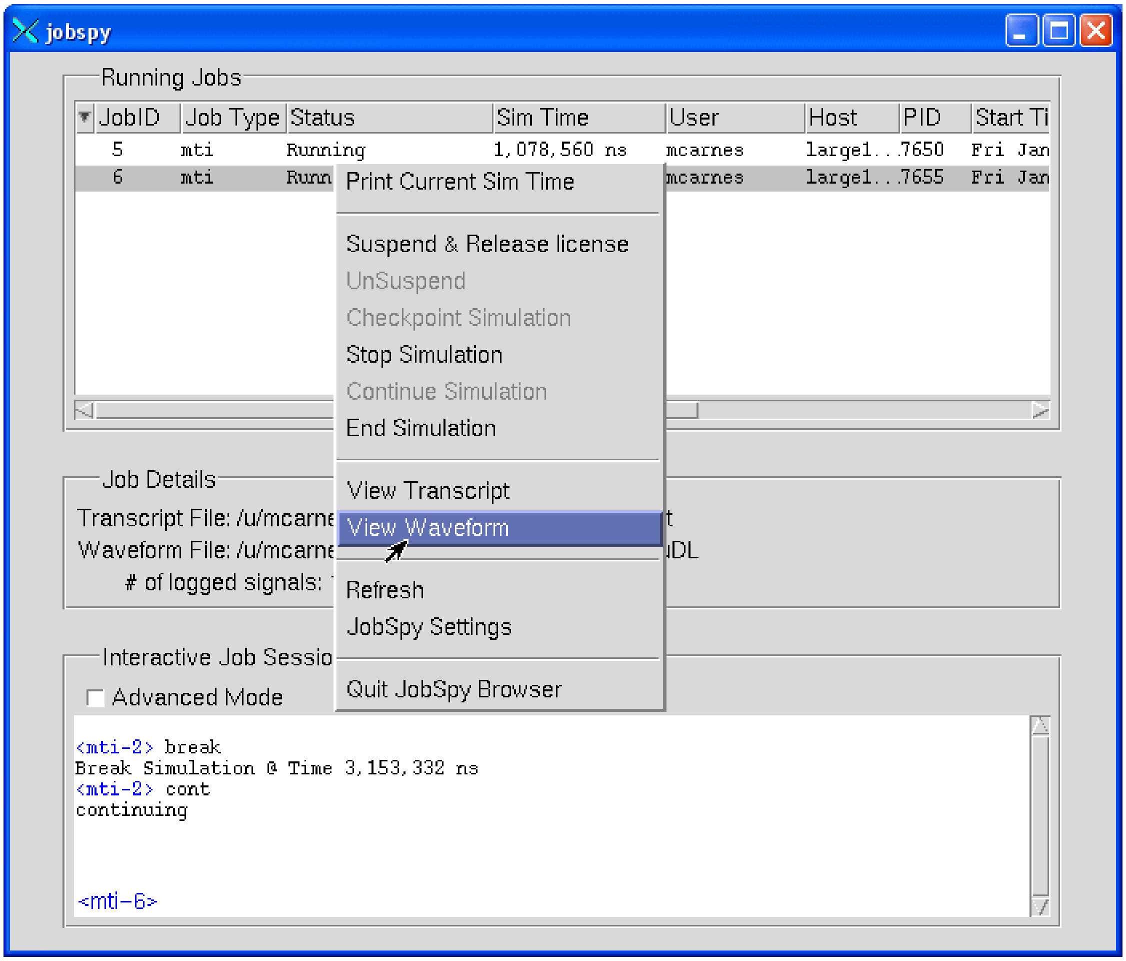
View Commands and Pathnames

The **View Transcript** and **View Waveform** commands read files (*transcript* and *<name>.wlf*, respectively) that are output by the simulator. These commands use the pathname from the remote machine to locate the required file. Depending on how your network is organized, the pathname may be different or inaccessible from the machine which is running JobSpy. In such cases, these commands will not work. The work around is to use **jobspy savewlf** to specify a known location for the WLF file or **cp** to copy the Transcript file to a known location.

Viewing Results During Active Simulation

You may want to check simulation results while your simulation jobs are still running. You can do this via the GUI or by using the **savewlf** command.

To view waveforms from the JobSpy GUI, right-click a job and select **View Waveform**.



JobID	Job Type	Status	Sim Time	User	Host	PID	Start Ti
5	mti	Running	1,078,560 ns	mcarnes	large1..	.7650	Fri Jan
6	mti	Runn		mcarnes	large1..	.7655	Fri Jan

Transcript File: /u/mcarne
Waveform File: /u/mcarne
# of logged signals:

☐ Advanced Mode

```
<mti-2> break
Break Simulation @ Time 3,153,332 ns
<mti-2> cont
continuing


<mti-6>
```

Here are two important points to remember about viewing waveforms from the GUI:

- You must first log signals before you can view them as waveforms. If you haven't logged any signals, the View Waveform command in the GUI will be disabled.
- View Waveform uses the pathname from the remote machine to access a WLF file. The command may not work on some networks. See View Commands and Pathnames for details.

Viewing Waveforms from the Command Line

From the command line, there are three steps to viewing waveforms:

1. Log the appropriate signals

   ```
   add log *
   ```

2. Save a dataset

   ```
   $ jobspy 1204 savewlf snap.wlf
   Dataset "sim" exported as WLF file: snap.wlf. @ 84,785,547 ns
   ```

3. View the dataset

   ```
   vsim -view snap.wlf
   ```

Licensing and Job Suspension

When you suspend a job via JobSpy, the simulation license is released by default. You can change this behavior by modifying the MTI_RELEASE_ON_SUSPEND environment variable. By default the variable is set to 10 (in seconds), which releases the license 10 seconds after receiving a suspend signal. If you change the value to 0 (off), simulation licenses will not be released upon job suspension.

Checkpointing Jobs

Checkpointing allows you to save the state of a simulation and restore it at a later time. There are three primary reasons for checkpointing jobs:

- Free up a license for a more important job
- Migrate a job from one machine to another
- Backup a job in case of a hardware crash or failure

In the case of freeing up a license, you should use the suspend command instead. Job suspension does not have the restrictions that checkpointing does.

Note

If you need to checkpoint a job for migration or backup, keep in mind the following restrictions:

The job must be restored on the same platform and exact OS on which the job was checkpointed.

If your job includes any foreign C code (i.e., PLI, FLI, etc.), the foreign application must be written to support checkpointing. See The PLI Callback reason Argument for more information on checkpointing with PLI applications. See the Foreign Language Interface Reference Manual for information on checkpointing with FLI applications.

Connecting to Load-Sharing Software

Load-sharing software such as Platform Computing's LSF™ or Sun's Grid Engine™ centralize management of distributed computing resources. JobSpy can access and monitor simulation runs that were submitted to these load-managing products.

With the exception of checkpointing (discussed below), the only requirement for connecting to load-sharing software is that the jobspy daemon be running prior to submitting the jobs. If the daemon is running, the jobs will show up in JobSpy automatically.

Checkpointing with Load-Sharing Software

Some additional steps are required to configure load-sharing software for checkpointing **vsim** jobs. The configuration depends on which load-sharing software you run.

Configuring LSF for Checkpointing

Do the following to enable **vsim** checkpointing with LSF:

- Set the environment variable LSB_ECHKPNT_METHOD_DIR to point to *<install_dir>/modeltech/<platform>*

  <platform> refers to the VCO for the ModelSim installation (e.g., linux, sunos5, etc.). See the *ModelSim Installation Guide* for a complete list.

- Set the environment variable LSB_ECHKPNT_METHOD to "modelsim"

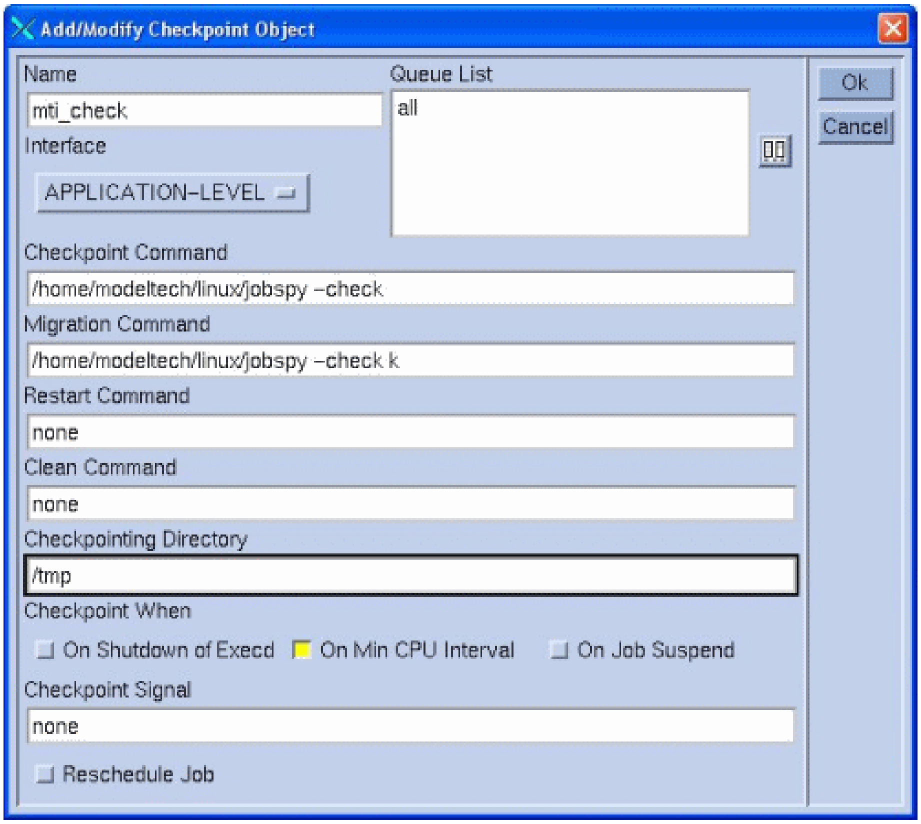
With these environment variables set, you can use standard LSF commands to checkpoint vsim jobs. Consult LSF documentation for information on those commands.

Configuring Flowtracer for Checkpointing

Flowtracer does not support checkpointing of **vsim** jobs.

Configuring Grid Engine for Checkpointing

To checkpoint **vsim** jobs with Grid Engine, you must create a Checkpoint Object. The dialog below shows an example configuration:



Name: mti_check

Queue List: all

Interface: APPLICATION-LEVEL

Checkpoint Command: /home/modeltech/linux/jobspy –check

Migration Command: /home/modeltech/linux/jobspy –check k

Restart Command: none

Clean Command: none

Checkpointing Directory: /tmp

Checkpoint When: □ On Shutdown of Execd ☑ On Min CPU Interval □ On Job Suspend

Checkpoint Signal: none

□ Reschedule Job

Note the following options in this dialog:

- The Interface should be set to APPLICAITON-LEVEL.
- The Checkpoint Command field should be set to *<install_dir>/modeltech/<platform>/jobspy -check*. <platform> refers to the VCO for the ModelSim installation (e.g., linux, sunos5, etc.). See the *ModelSim Installation Guide* for a complete list.
- The Migration Command field should be set to *<install_dir>/modeltech/<platform>/jobspy -check k*

Consult the Grid Engine documentation for additional information.

Chapter 20
Generating Stimulus with Waveform Editor

The ModelSim Waveform Editor offers a simple method for creating design stimulus. You can generate and edit waveforms in a graphical manner and then drive the simulation with those waveforms. With Waveform Editor you can do the following:

- Create waveforms using four predefined patterns: clock, random, repeater, and counter. See Creating Waveforms from Patterns.
- Edit waveforms with numerous functions including inserting, deleting, and stretching edges; mirroring, inverting, and copying waveform sections; and changing waveform values on-the-fly. See Editing Waveforms.
- Drive the simulation directly from the created waveforms
- Save created waveforms to four stimulus file formats: Tcl force format, extended VCD format, Verilog module, or VHDL architecture. The HDL formats include code that matches the created waveforms and can be used in testbenches to drive a simulation. See Exporting Waveforms to a Stimulus File

Limitations

The current version does not support the following:

- Enumerated signals, records, multi-dimensional arrays, and memories
- User-defined types
- SystemC or SystemVerilog

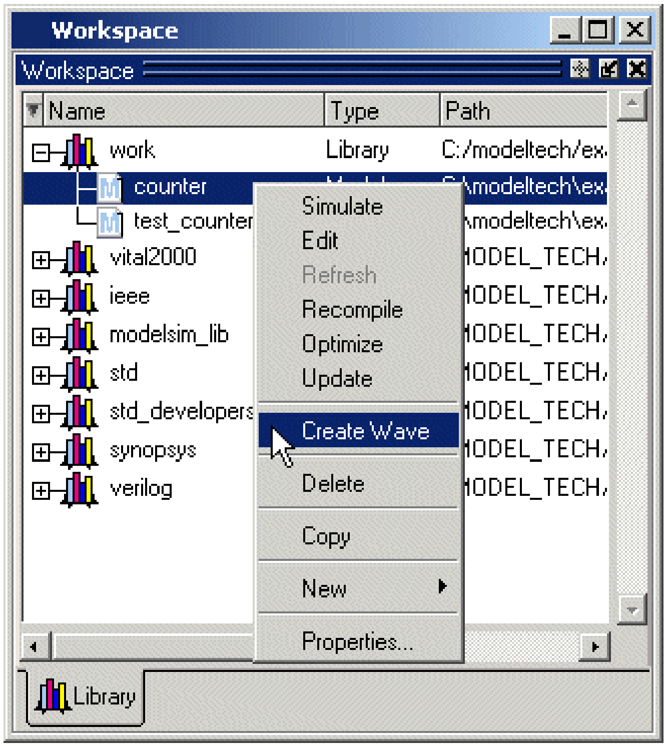
Getting Started with the Waveform Editor

You can use Waveform Editor before or after loading a design. Regardless of which method you choose, you will select design objects and use them as the basis for created waveforms.

Using Waveform Editor Prior to Loading a Design

Here are the basic steps for using waveform editor prior to loading a design:

1. Right-click a design unit on the Library tab of the Workspace pane and select Create Wave.

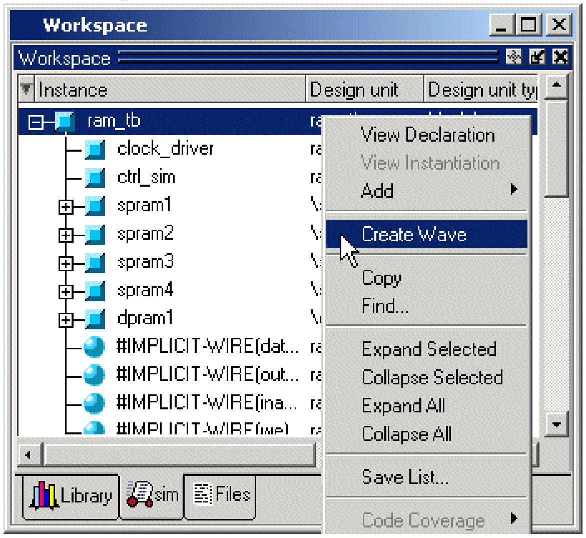


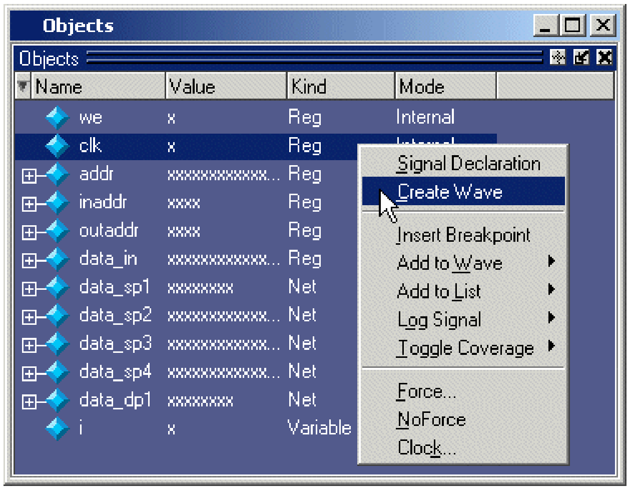
1. Edit the waveforms in the Wave window. See Editing Waveforms for more details.
2. Run the simulation (see Simulating Directly from Waveform Editor) or save the created waveforms to a stimulus file (see Exporting Waveforms to a Stimulus File).

Using Waveform Editor After Loading a Design

Here are the basic steps for using waveform editor after loading a design:

1. Right-click a block in the structure tab of the Workspace pane or an object in the Object pane and select Create Wave.





Name	Value	Kind	Mode
we	x	Reg	Internal
clk	x	Reg	Internal
addr	xxxxxxxxxxxx...	Reg	
inaddr	xxxx	Reg	
outaddr	xxxx	Reg	
data_in	xxxxxxxxxxxx...	Reg	
data_sp1	xxxxxxxx	Net	
data_sp2	xxxxxxxxxxxx...	Net	
data_sp3	xxxxxxxxxxxx...	Net	
data_sp4	xxxxxxxxxxxx...	Net	
data_dp1	xxxxxxxx	Net	
i	x	Variable	

2. Use the Create Pattern wizard to create the waveforms (see Creating Waveforms from Patterns).

3. Edit the waveforms as required (see Editing Waveforms).

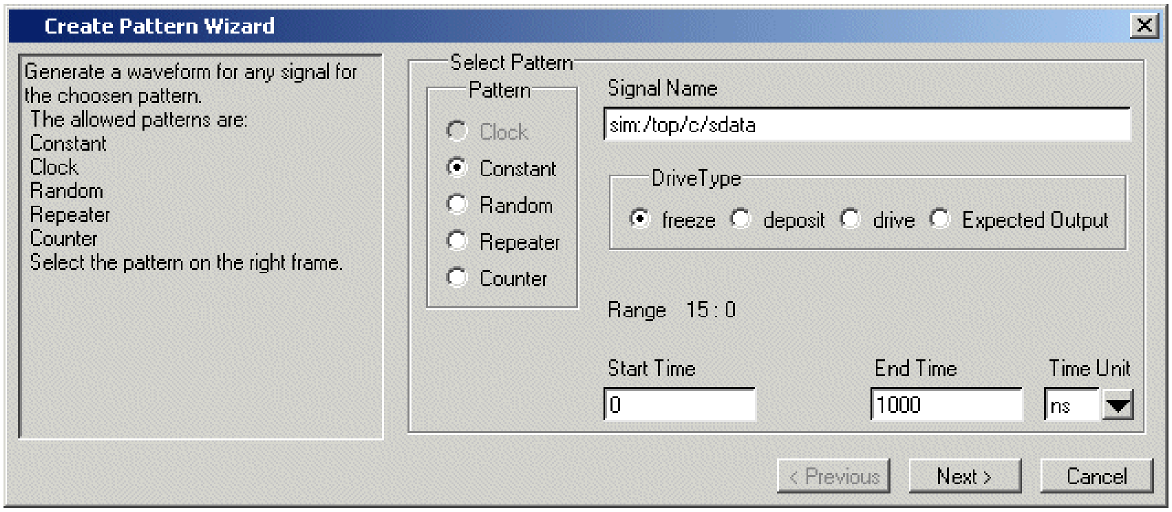
4. Run the simulation (see Simulating Directly from Waveform Editor) or save the created waveforms to a stimulus file (see Exporting Waveforms to a Stimulus File).

Creating Waveforms from Patterns

Waveform Editor includes a Create Pattern wizard that walks you through the process of creating waveforms. To access the wizard:

- Right-click an object in the Objects pane or structure pane and select Create Wave.
- Right-click a signal already in the Wave window and select Create/Modify Waveform. (Only possible before simulation is run.)

The graphic below shows the initial dialog in the wizard. Note that the Drive Type field is not present for input and output signals.



Create Pattern Wizard

Generate a waveform for any signal for the choosen pattern.
The allowed patterns are:
Constant
Clock
Random
Repeater
Counter
Select the pattern on the right frame.

Select Pattern

Pattern
○ Clock
● Constant
○ Random
○ Repeater
○ Counter

Signal Name: sim:/top/c/sdata

DriveType: ● freeze ○ deposit ○ drive ○ Expected Output

Range 15 : 0

Start Time: 0

End Time: 1000

Time Unit: ns

In this dialog you specify the signal that the waveform will be based upon, the Drive Type (if applicable), the start and end time for the waveform, and the pattern for the waveform.

The second dialog in the wizard lets you specify the appropriate attributes based on the pattern you select. The table below shows the five available patterns and their attributes:

Table 20-1.

Pattern	Description
Clock	Specify an initial value, duty cycle, and clock period for the waveform.
Constant	Specify a value.
Random	Generates different patterns depending upon the seed value. Specify the type (normal or uniform), an initial value, and a seed value. If you don't specify a seed value, ModelSim uses a default value of 5.
Repeater	Specify an initial value and pattern that repeats. You can also specify how many times the pattern repeats.
Counter	Specify start and end values, time period, type (Range, Binary, Gray, One Hot, Zero Hot, Johnson), counter direction, step count, and repeat number.

Editing Waveforms

You can edit waveforms interactively with menu commands, mouse actions, or by using the wave edit command.

To edit waveforms in the Wave window, follow these steps:

1. Create an editable pattern as described under Creating Waveforms from Patterns.
2. Enter editing mode by selecting **View > Zoom > Mouse Mode > Edit Mode** or by clicking the Edit Mode icon.
3. Select an edge or a section of the waveform with your mouse. See Selecting Parts of the Waveform for more details.
4. Select a command from the **Edit > Edit Wave** menu or right-click on the waveform and select a command from the Edit Wave context menu.

The table below summarizes the editing commands that are available.

Table 20-2.

Operation	**Description**
Cut	Cut the selected portion of the waveform to the clipboard
Copy	Copy the selected portion of the waveform to the clipboard
Value	Change the value of the selected portion of the waveform
Delete Edge	Delete the edge at the active cursor
Insert Pulse	Insert a pulse at the location of the active cursor
Invert	Invert the selected waveform section
Mirror	Mirror the selected waveform section
Paste	Paste the contents of the clipboard over the selected section or at the active cursor location
Stretch Edge	Move an edge forward/backward by "stretching" the waveform; see Stretching and Moving Edges for more information
Move Edge	Move an edge forward/backward without changing other edges; see Stretching and Moving Edges for more information
Drive Type	Change the drive type of the selected portion of the waveform
Extend All Waves	Extend all created waveforms by the specified amount or to the specified simulation time; ModelSim cannot undo this edit or any edits done prior to an extend command
Undo	Undo waveform edits (except changing drive type and extending all waves)
Redo	Redo previously undone waveform edits

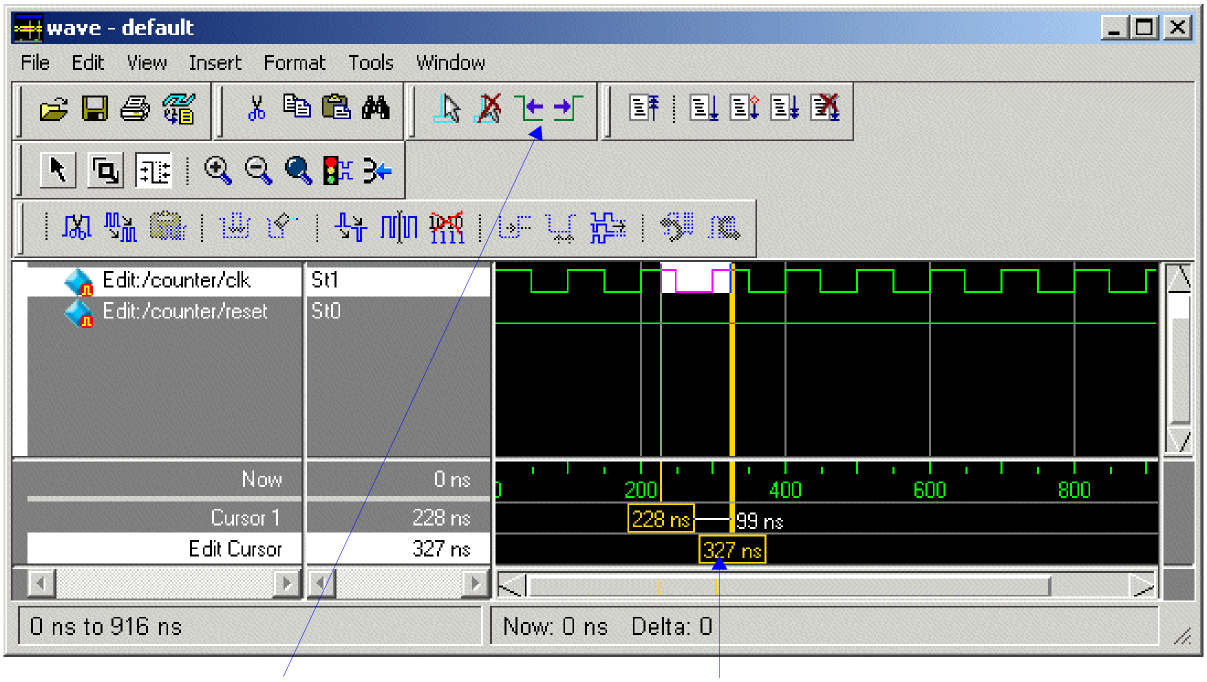
These commands can also be accessed via toolbar buttons. See Waveform Editor Toolbar for more information.

Selecting Parts of the Waveform

There are several methods for selecting edges or sections of a waveform. The table and graphic below describe the various options.

Table 20-3.

Action	Method
Select a waveform edge	Click on or just to the right of the waveform edge
Select a section of the waveform	Click-and-drag the mouse pointer in the waveform pane
Select a section of multiple waveforms	Click-and-drag the mouse pointer while holding the <Shift> key
Extend/contract the selection size	Drag a cursor in the cursor pane
Extend/contract selection from edge-to-edge	Click Next Transition/Previous Transition icons after selecting section



Click these icons to extend/contract selection from edge-to-edge

Drag cursor here to extend/contract selection

Selection and Zoom Percentage

You may find that you cannot select the exact range you want because the mouse moves more than one unit of simulation time (e.g., 228 ns to 230 ns). If this happens, zoom in on the Wave display (see Zooming the Wave Window Display), and you should be able to select the range you want.

Auto Snapping of the Cursor

When you click just to the right of a waveform edge in the waveform pane, the cursor automatically "snaps" to the nearest edge. This behavior is controlled by the Snap Distance setting in the Wave window preferences dialog.

Stretching and Moving Edges

There are mouse and keyboard shortcuts for moving and stretching edges:

Table 20-4.

Action	Mouse/keyboard shortcut
Stretch an edge	Hold the <Ctrl> key and drag the edge
Move an edge	Hold the <Ctrl> key and drag the edge with the 2nd (middle) mouse button

Here are some points to keep in mind about stretching and moving edges:

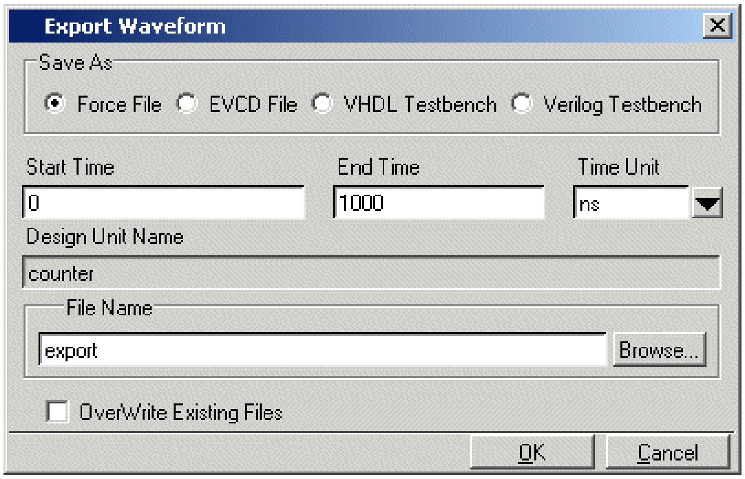
- If you stretch an edge forward, more waveform is inserted at the beginning of simulation time.
- If you stretch an edge backward, waveform is deleted at the beginning of simulation time.
- If you move an edge past another edge, either forward or backward, the edge you moved past is deleted.

Simulating Directly from Waveform Editor

You need not save the waveforms in order to use them as stimulus for a simulation. Once you have configured all the waveforms, you can run the simulation as normal by selecting **Simulate > Start Simulation** in the Main window or using the vsim command. ModelSim automatically uses the created waveforms as stimulus for the simulation. Furthermore, while running the simulation you can continue editing the waveforms to modify the stimulus for the part of the simulation yet to be completed.

Exporting Waveforms to a Stimulus File

Once you have created and edited the waveforms, you can save the data to a stimulus file that can be used to drive a simulation now or at a later time. To save the waveform data, select **File > Export Waveform** or use the wave export command.



Export Waveform

Save As
● Force File ○ EVCD File ○ VHDL Testbench ○ Verilog Testbench

Start Time: 0
End Time: 1000
Time Unit: ns

Design Unit Name: counter

File Name: export

□ OverWrite Existing Files

You can save the waveforms in four different formats:

Table 20-5.

Format	**Description**
Force format	Creates a Tcl script that contains force commands necessary to recreate the waveforms; source the file when loading the simulation as described under Driving Simulation with the Saved Stimulus File
EVCD format	Creates an extended VCD file which can be reloaded using the **Import > EVCD File** command or can be used with the **-vcdstim** argument to vsim to simulate the design
VHDL Testbench	Creates a VHDL architecture that you load as the top-level design unit
Verilog Testbench	Creates a Verilog module that you load as the top-level design unit

Driving Simulation with the Saved Stimulus File

The method for loading the stimulus file depends upon what type of format you saved. In each of the following examples, assume that the top-level of your block is named "top" and you saved the waveforms to a stimulus file named "mywaves" with the default extension.

Table 20-6.

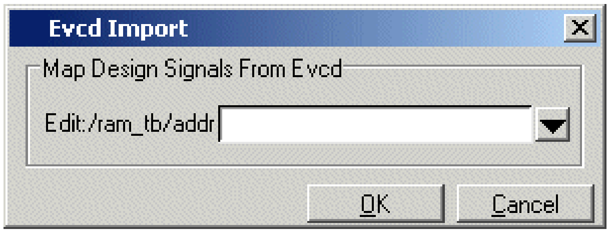
Format	Loading example
Force format	vsim top -do mywaves.do
Extended VCD format[1]	vsim top -vcdstim mywaves.vcd
VHDL Testbench	vcom mywaves.vhd vsim mywaves
Verilog Testbench	vlog mywaves.v vsim mywaves

[1] You can also use the **Import > EVCD** command from the Wave window. See below for more details on working with EVCD files.

Signal Mapping and Importing EVCD Files

When you import a previously saved EVCD file, ModelSim attempts to map the signals in the EVCD file to the signals in the loaded design. It matches signals based on name and width.

If ModelSim can't map the signals automatically, you can do the mapping yourself by selecting one or more signals, right-clicking a selected signal, and then selecting Map to Design Signal.



Select a signal from the drop-down arrow and click OK. You will repeat this process for each signal you selected.

Note

This command works only with extended VCD files created with ModelSim.

Using Waveform Compare with Created Waveforms

The Waveform Compare feature compares two or more waveforms and displays the differences in the Wave window (see Waveform Compare for details). This feature can be used in tandem with Waveform Editor. The combination is most useful in situations where you know the expected output of a signal and want to compare visually the differences between expected output and simulated output.

The basic procedure for using the two features together is as follows:

- Create a waveform based on the signal of interest with a drive type of expected output
- Add the design signal of interest to the Wave window and then run the design
- Start a comparison and use the created waveform as the reference dataset for the comparison. Use the text "Edit" to designate a create waveform as the reference dataset. For example:

```
compare start Edit sim
compare add -wave /test_counter/count
compare run
```

Saving the Waveform Editor Commands

When you create and edit waveforms in the Wave window, ModelSim tracks the underlying Tcl commands and reports them to the transcript. You can save those commands to a DO file that can be run at a later time to recreate the waveforms.

To save your waveform editor commands, select **File** > **Save**.

Chapter 21
Standard Delay Format (SDF) Timing Annotation

This chapter discusses ModelSim's implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data can be annotated from SDF files by using the simulator's built-in SDF annotator.

ASIC and FPGA vendors usually provide tools that create SDF files for use with their cell libraries. Refer to your vendor's documentation for details on creating SDF files for your library. Many vendors also provide instructions on using their SDF files and libraries with ModelSim.

The SDF specification was originally created for Verilog designs, but it has also been adopted for VHDL VITAL designs. In general, the designer does not need to be familiar with the details of the SDF specification because the cell library provider has already supplied tools that create SDF files that match their libraries.

Note

ModelSim will read SDF files that were compressed using gzip. Other compression formats (e.g., Unix zip) are not supported.

Specifying SDF Files for Simulation

ModelSim supports SDF versions 1.0 through 4.0 (except the NETDELAY statement). The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following vsim command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

Instance Specification

The instance paths in the SDF file are relative to the instance to which the SDF is applied. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

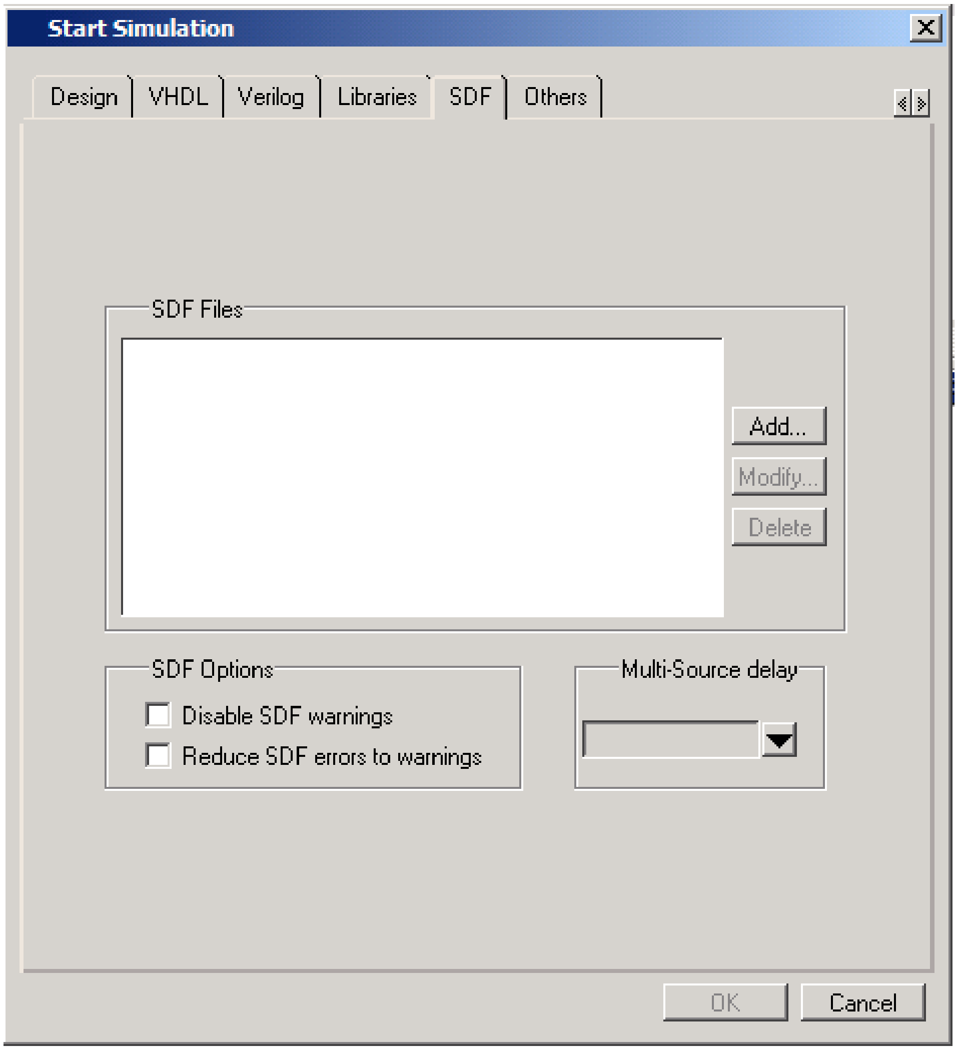
```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design can have several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

SDF Specification with the GUI

As an alternative to the command-line options, you can specify SDF files in the **Start Simulation** dialog box under the SDF tab.



☐ Disable SDF warnings
☐ Reduce SDF errors to warnings

You can access this dialog by invoking the simulator without any arguments or by selecting **Simulate** > **Start Simulation**. See the GUI chapter for a description of this dialog.

For Verilog designs, you can also specify SDF files by using the **$sdf_annotate** system task. See $sdf_annotate for more details.

Errors and Warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not. Use the **-sdfnoerror** option with vsim to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using vsim with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** tab from the **Start Simulation** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn +nosdfwarn) to disable warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See Troubleshooting for more information on errors and warnings and how to avoid them.

Compiling SDF Files

The sdfcom command compiles SDF files. In situations where the same SDF file is used for multiple simulation runs, the elaboration time will be reduced significantly. Depending on the design, time savings of 25% to 60% may be realized.

In the current release, compiled SDF is supported for purely Verilog regions only. For example, if a VHDL testbench instantiates a Verilog module *dut*, and all sub-instances of *dut* are Verilog, then you may annotate a compiled SDF file for the region under *dut*. If the annotated region is not pure Verilog, ModelSim will issue an error message.

Note

When compiled SDF files are used, the annotator behaves as if the -v2k_int_delays switch for the vsim command has been specified.

Simulating with Compiled SDF Files

Compiled SDF files may be specified on the vsim line with the **-sdfmin**, **-sdftyp**, and **-sdfmax** arguments. Alternatively, they may be specified as the filename in a $sdf_annotate() system task in the Verilog source.

Using $sdf_annotate() with Compiled SDF

The following limitations exist when using compiled SDF files with $sdf_annotate():

- The $sdf_annotate() call cannot be made from a delayed initial block:

```
initial #10 $sdf_annotate(...); // Not allowed
```

- The $sdf_annotate() call cannot be made from an if statement:

```
reg doSdf = 1'b1;
initial begin
  if (doSdf) $sdf_annotate(...); // Not allowed
end
```

- 3. Within a "begin..end" block, a $sdf_annotate() call cannot be preceded by anything other than another $sdf_annotate() call:

```
initial begin
  $sdf_annotate("sdffile1", top);
  $display("Annotated 'sdffile1'");
  $sdf_annotate("sdffile2", top); // Not allowed
end
```

- If the annotation order of multiple $sdf_annotate() calls is important, you must have all of them in a single initial block.

VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary may help you understand simulator error messages. For additional VITAL specification information, see VITAL Specification and Source Code.

SDF to VHDL Generic Matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

Table 21-1.

SDF construct	Matching VHDL generic name
(IOPATH a y (3))	tpd_a_y
(IOPATH (posedge clk) q (1) (2))	tpd_clk_q_posedge
(INTERCONNECT u1/y u2/a (5))	tipd_a
(SETUP d (posedge clk) (5))	tsetup_d_clk_noedge_posedge
(HOLD (negedge d) (posedge clk) (5))	thold_d_clk_negedge_posedge
(SETUPHOLD d clk (5) (5))	tsetup_d_clk & thold_d_clk
(WIDTH (COND (reset==1’b0) clk) (5))	tpw_clk_reset_eq_0

The SDF statement CONDELSE, when targeted for Vital cells, is annotated to a **tpd** generic of the form **tpd_<inputPort>_<outputPort>**.

Resolving Errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
** Error (vsim-SDF-3240) myasic.sdf(18):
Instance ’/testbench/dut/u1’ does not have a generic named ’tpd_a_y’
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files provided by the cell library vendor.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don’t match the names expected by the annotator, then there are several possibilities:

- The vendor’s tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor’s library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke vsim with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see Troubleshooting.

Verilog SDF

Verilog designs can be annotated using either the simulator command-line options or the **$sdf_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **$sdf_annotate** task annotates the design at the time it is called in the Verilog source code. This provides more flexibility than the command-line options.

$sdf_annotate

Syntax

```
$sdf_annotate
    (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"], ["<mtm_spec>"],
    ["<scale_factor>"], ["<scale_type>"]);
```

Arguments

- "<sdffile>"

  String that specifies the SDF file. Required.

- <instance>

  Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the $sdf_annotate call is made.

- "<config_file>"

  String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

- "<log_file>"

  String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

- "<mtm_spec>"

  String that specifies the delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is "tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).

- "<scale_factor>"

  String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

- "<scale_type>"

  String that overrides the **<mtm_spec>** delay selection. Optional. The **<mtm_spec>** delay selection is always used to select the delay scaling factor, but if a **<scale_type>** is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the **<mtm_spec>** value.

Examples

Optional arguments can be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance to which it applies:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

SDF to Verilog Construct Matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct can have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

- **IOPATH** is matched to specify path delays or primitives:

SDF	Verilog
(IOPATH (posedge clk) q (3) (4))	(posedge clk => q) = 0;
(IOPATH a y (3) (4))	buf u1 (y, a);

  The IOPATH construct usually annotates path delays. If ModelSim can’t locate a corresponding specify path delay, it returns an error unless you use the **+sdf_iopath_to_prim_ok** argument to vsim. If you specify that argument and the module contains no path delays, then all primitives that drive the specified output port are annotated.

- **INTERCONNECT** and **PORT** are matched to input ports:

SDF	Verilog
(INTERCONNECT u1.y u2.a (5))	input a;
(PORT u2.a (5))	inout a;

  Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

- **PATHPULSE** and **GLOBALPATHPULSE** are matched to specify path delays:

SDF	Verilog
(PATHPULSE a y (5) (10))	(a => y) = 0;
(GLOBALPATHPULSE a y (30) (60))	(a => y) = 0;

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

- **DEVICE** is matched to primitives or specify path delays:

SDF	Verilog
(DEVICE y (5))	and u1(y, a, b);
(DEVICE y (5))	(a => y) = 0; (b => y) = 0;

If the SDF cell instance is a primitive instance, then that primitive's delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

SETUP is matched to $setup and $setuphold:

SDF	Verilog
(SETUP d (posedge clk) (5))	$setup(d, posedge clk, 0);
(SETUP d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

- **HOLD** is matched to $hold and $setuphold:

SDF	Verilog
(HOLD d (posedge clk) (5))	$hold(posedge clk, d, 0);
(HOLD d (posedge clk) (5))	$setuphold(posedge clk, d, 0, 0);

- **SETUPHOLD** is matched to $setup, $hold, and $setuphold:

SDF	Verilog
(SETUPHOLD d (posedge clk) (5) (5))	$setup(d, posedge clk, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$hold(posedge clk, d, 0);
(SETUPHOLD d (posedge clk) (5) (5))	$setuphold(posedge clk, d, 0, 0);

- **RECOVERY** is matched to $recovery:

SDF	Verilog
(RECOVERY (negedge reset) (posedge clk) (5))	$recovery(negedge reset, posedge clk, 0);

- **REMOVAL** is matched to $removal:

SDF	Verilog
(REMOVAL (negedge reset) (posedge clk) (5))	$removal(negedge reset, posedge clk, 0);

- **RECREM** is matched to $recovery, $removal, and $recrem:

SDF	Verilog
(RECREM (negedge reset) (posedge clk) (5) (5))	$recovery(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$removal(negedge reset, posedge clk, 0);
(RECREM (negedge reset) (posedge clk) (5) (5))	$recrem(negedge reset, posedge clk, 0);

- **SKEW** is matched to $skew:

SDF	Verilog
(SKEW (posedge clk1) (posedge clk2) (5))	$skew(posedge clk1, posedge clk2, 0);

- **WIDTH** is matched to $width:

SDF	Verilog
(WIDTH (posedge clk) (5))	$width(posedge clk, 0);

- **PERIOD** is matched to $period:

SDF	Verilog
(PERIOD (posedge clk) (5))	$period(posedge clk, 0);

- **NOCHANGE** is matched to $nochange:

SDF	Verilog
(NOCHANGE (negedge write) addr (5) (5))	$nochange(negedge write, addr, 0, 0);

Optional Edge Specifications

Timing check ports and path delay input ports can have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(posedge data, posedge clk, 0);
(SETUP data (posedge clock) (5))	$setup(negedge data, posedge clk, 0);

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value.

Likewise, the SDF file may contain more accurate data than the model can accommodate.

SDF	Verilog
(SETUP (posedge data) (posedge clock) (4))	$setup(data, posedge clk, 0);
(SETUP (negedge data) (posedge clock) (6))	$setup(data, posedge clk, 0);

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers can also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. For example,

SDF	Verilog
(SETUP data (posedge clock) (5))	$setup(data, edge[01, 0x] clk, 0);

The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port.

Optional Conditions

Timing check ports and path delays can have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

```
SDF                                   Verilog

(SETUP data (COND (reset!=1)          $setup(data, posedge clk &&&
    (posedge clock)) (5))                 (reset==0),0);
```

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

```
SDF                                   Verilog

(COND (r1 || r2) (IOPATH clk q (5)))  if (r1 || r2) (clk => q) = 5; // matches

(COND (r1 || r2) (IOPATH clk q (5)))  if (r2 || r1) (clk => q) = 5; // does not match
```

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

Rounded Timing Values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells can be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog $sdf_annotate system task can annotate Verilog cells only. See the vsim command for more information on SDF command-line options.

Interconnect Delays

An interconnect delay represents the delay from the output of one device to the input of another. ModelSim can model single interconnect delays or multisource interconnect delays for Verilog, VHDL/VITAL, or mixed designs. See the **vsim** command for more information on the relevant command-line arguments.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations, because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Disabling Timing Checks

ModelSim offers a number of options for disabling timing checks on a "global" or individual basis. The table below provides a summary of those options. See the command and argument descriptions in the Reference Manual for more details.

Table 21-2.

Command and argument	**Effect**
tcheck_set	modifies reporting or X generation status on one or more timing checks
tcheck_status	prints to the Transcript the current status of one or more timing checks
vlog +notimingchecks	disables timing check system tasks for all instances in the specified Verilog design
vlog +nospecify	disables specify path delays and timing checks for all instances in the specified Verilog design
vsim +no_neg_tchk	disables negative timing check limits by setting them to zero for all instances in the specified design
vsim +no_notifier	disables the toggling of the notifier register argument of the timing check system tasks for all instances in the specified design
vsim +no_tchk_msg	disables error messages issued by timing check system tasks when timing check violations occur for all instances in the specified design
vsim +notimingchecks	disables Verilog and VITAL timing checks for all instances in the specified design
vsim +nospecify	disables specify path delays and timing checks for all instances in the specified design

Troubleshooting

Specifying the Wrong Instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator’s SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is generally wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. See Instance Specification for an example.

A common example for both VHDL and Verilog testbenches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

VHDL Testbench

```
entity testbench is end;
architecture only of testbench is
   component myasic
   end component;
begin
   dut : myasic;
end;
```

Verilog Testbench

```
module testbench;
   myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you can leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important thing is to select the instance for which the SDF is intended. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, view the structure pane, navigate to the model instance, select it, and enter the environment command. This command displays the instance name that should be used in the SDF command-line option.

Mistaking a Component or Module Name for an Instance Label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/myasic’.
```

Forgetting to Specify the Instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

vsim -sdfmax myasic.sdf testbench

Results in:

```
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u1’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u2’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u3’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u4’
** Error (vsim-SDF-3250) myasic.sdf(0):
Failed to find INSTANCE ’/testbench/u5’
** Warning (vsim-SDF-3432) myasic.sdf:
This file is probably applied to the wrong instance.
** Warning (vsim-SDF-3432) myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
** Warning (vsim-SDF-3440) myasic.sdf:
Failed to find any of the 358 instances from this file.
** Warning (vsim-SDF-3442) myasic.sdf:
Try instance ’/testbench/dut’. It contains all instance paths from this
file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see Resolving Errors for specific VHDL VITAL SDF troubleshooting.

Chapter 22
Value Change Dump (VCD) Files

This chapter describes how to use VCD files in ModelSim. The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes.

VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. ModelSim provides command equivalents for these system tasks and extends VCD support to VHDL designs. The ModelSim commands can be used on VHDL, Verilog, or mixed designs.

If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

Creating a VCD File

There are two flows in ModelSim for creating a VCD file. One flow produces a four-state VCD file with variable changes in 0, 1, x, and z with no strength information; the other produces an extended VCD file with variable changes in all states and strength information and port driver data.

Both flows will also capture port driver changes unless filtered out with optional command-line arguments.

Flow for Four-State VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name with the vcd file command and add objects to the file with the vcd add command:

```
VSIM 1> vcd file myvcdfile.vcd
VSIM 2> vcd add /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be a VCD file in the working directory.

Flow for Extended VCD File

First, compile and load the design:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
```

Next, with the design loaded, specify the VCD file name and objects to add with the vcd dumpports command:

```
VSIM 1> vcd dumpports -file myvcdfile.vcd /test_counter/dut/*
VSIM 3> run
VSIM 4> quit -f
```

There will now be an extended VCD file in the working directory.

By default ModelSim uses strength ranges for resolving conflicts as specified by IEEE 1364-2005. You can ignore strength ranges using the **-no_strength_range** argument to the vcd dumpports command. See Resolving Values for more details.

Case Sensitivity

VHDL is not case sensitive so ModelSim converts all signal names to lower case when it produces a VCD file. Conversely, Verilog designs are case sensitive so ModelSim maintains case when it produces a VCD file.

Checkpoint/Restore and Writing VCD Files

If a checkpoint occurs while ModelSim is writing a VCD file, the entire VCD file is copied into the checkpoint file. Since VCD files can be very large, it is possible that disk space problems may occur. Consequently, ModelSim issues a warning in this situation.

Using Extended VCD as Stimulus

You can use an extended VCD file as stimulus to re-simulate your design. There are two ways to do this: 1) simulate the top level of a design unit with the input values from an extended VCD file; and 2) specify one or more instances in a design to be replaced with the output values from the associated VCD file.

Simulating with Input Values from a VCD File

When simulating with inputs from an extended VCD file, you can simulate only one design unit at a time. In other words, you can apply the VCD file inputs only to the top level of the design unit for which you captured port data.

The general procedure includes two steps:

1. Create a VCD file for a single design unit using the vcd dumpports command.
2. Resimulate the single design unit using the **-vcdstim** argument to vsim. Note that **-vcdstim** works only with VCD files that were created by a ModelSim simulation.

Example 22-1. Verilog Counter

First, create the VCD file for the single instance using **vcd dumpports**:

```
% cd ~/modeltech/examples/misc
% vlib work
% vlog counter.v tcounter.v
% vsim test_counter
VSIM 1> vcd dumpports -file counter.vcd /test_counter/dut/*
VSIM 2> run
VSIM 3> quit -f
```

Next, rerun the counter without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim counter.vcd counter
VSIM 1> add wave /*
VSIM 2> run 200
```

Example 22-2. VHDL Adder

First, create the VCD file using **vcd dumpports**:

```
% cd ~/modeltech/examples/misc
% vlib work
% vcom gates.vhd adder.vhd stimulus.vhd
% vsim testbench2
VSIM 1> vcd dumpports -file addern.vcd /testbench2/uut/*
VSIM 2> run 1000
VSIM 3> quit -f
```

Next, rerun the adder without the testbench, using the **-vcdstim** argument:

```
% vsim -vcdstim addern.vcd addern -gn=8 -do "add wave /*; run 1000"
```

Example 22-3. Mixed-HDL Design

First, create three VCD files, one for each module:

```
% cd ~/modeltech/examples/tutorials/mixed/projects
% vlib work
% vlog cache.v memory.v proc.v
% vcom util.vhd set.vhd top.vhd
% vsim top
VSIM 1> vcd dumpports -file proc.vcd /top/p/*
VSIM 2> vcd dumpports -file cache.vcd /top/c/*
VSIM 3> vcd dumpports -file memory.vcd /top/m/*
VSIM 4> run 1000
VSIM 5> quit -f
```

Next, rerun each module separately, using the captured VCD stimulus:

```
% vsim -vcdstim proc.vcd proc -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim cache.vcd cache -do "add wave /*; run 1000"
VSIM 1> quit -f

% vsim -vcdstim memory.vcd memory -do "add wave /*; run 1000"
VSIM 1> quit -f
```

Replacing Instances with Output Values from a VCD File

Replacing instances with output values from a VCD file lets you simulate without the instance's source or even the compiled object. The general procedure includes two steps:

1. Create VCD files for one or more instances in your design using the vcd dumpports command. If necessary, use the **-vcdstim** switch to handle port order problems (see below).
2. Re-simulate your design using the **-vcdstim <instance>=<filename>** argument to vsim. Note that this works only with VCD files that were created by a ModelSim simulation.

Example 22-4. Replacing Instances

In the following example, the three instances */top/p*, */top/c*, and */top/m* are replaced in simulation by the output values found in the corresponding VCD files.

First, create VCD files for all instances you want to replace:

```
vcd dumpports -vcdstim -file proc.vcd /top/p/*
vcd dumpports -vcdstim -file cache.vcd /top/c/*
vcd dumpports -vcdstim -file memory.vcd /top/m/*
run 1000
```

Next, simulate your design and map the instances to the VCD files you created:

```
vsim top -vcdstim /top/p=proc.vcd -vcdstim /top/c=cache.vcd
-vcdstim /top/m=memory.vcd
```

Port Order Issues

The **-vcdstim** argument to the **vcd dumpports** command ensures the order that port names appear in the VCD file matches the order that they are declared in the instance's module or entity declaration. Consider the following module declaration:

```
module proc(clk, addr, data, rw, strb, rdy);
  input  clk, rdy;
  output addr, rw, strb;
  inout  data;
```

The order of the ports in the module line (clk, addr, data, ...) does not match the order of those ports in the input, output, and inout lines (clk, rdy, addr, ...). In this case the **-vcdstim** argument to the **vcd dumpports** command needs to be used.

In cases where the order is the same, you do not need to use the **-vcdstim** argument to **vcd dumpports**. Also, module declarations of the form:

```
module proc(input clk, output addr, inout data, ...)
```

do not require use of the argument.

VCD Commands and VCD Tasks

ModelSim VCD commands map to IEEE Std 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below maps the VCD commands to their associated tasks.

Table 22-1.

VCD commands	VCD system tasks
vcd add	$dumpvars
vcd checkpoint	$dumpall
vcd file	$dumpfile
vcd flush	$dumpflush
vcd limit	$dumplimit
vcd off	$dumpoff
vcd on	$dumpon

ModelSim also supports extended VCD (dumpports system tasks). The table below maps the VCD dumpports commands to their associated tasks.

Table 22-2.

VCD dumpports commands	VCD system tasks
vcd dumpports	$dumpports
vcd dumpportsall	$dumpportsall
vcd dumpportsflush	$dumpportsflush
vcd dumpportslimit	$dumpportslimit
vcd dumpportsoff	$dumpportsoff
vcd dumpportson	$dumpportson

ModelSim supports multiple VCD files. This functionality is an extension of the IEEE Std 1364 specification. The tasks behave the same as the IEEE equivalent tasks such as $dumpfile, $dumpvar, etc. The difference is that $fdumpfile can be called multiple times to create more than one VCD file, and the remaining tasks require a filename argument to associate their actions with a specific file.

Table 22-3.

VCD commands	VCD system tasks
vcd add -file <filename>	$fdumpvars
vcd checkpoint <filename>	$fdumpall
vcd files <filename>	$fdumpfile
vcd flush <filename>	$fdumpflush
vcd limit <filename>	$fdumplimit
vcd off <filename>	$fdumpoff
vcd on <filename>	$fdumpon

Compressing Files with VCD Tasks

ModelSim can produce compressed VCD files using the **gzip** compression algorithm. Since we cannot change the syntax of the system tasks, we act on the extension of the output file name. If you specify a *.gz* extension on the filename, ModelSim will compress the output.

VCD File from Source To Output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

VHDL Source Code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
   port (CLK, RESET, data_in   : IN STD_LOGIC;
      Q : INOUT STD_LOGIC_VECTOR(8 downto 0));
END SHIFTER_MOD ;

architecture RTL of SHIFTER_MOD is
begin
   process (CLK,RESET)
   begin
      if (RESET = '1') then
         Q <= (others => '0') ;
      elsif (CLK'event and CLK = '1') then
         Q <= Q(Q'left - 1 downto 0) & data_in ;
      end if ;
   end process ;
end ;
```

VCD Simulator Commands

At simulator time zero, the designer executes the following commands:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0
force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
quit -sim
```

VCD Output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

```
$date
   Thu Sep 18 11:07:43 2003
$end
$version
   ModelSim Version 6.1
$end
$timescale
   1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ' q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpoff
x!
x"
x#
x$
x%
x&
x'
x(
x)
x*
x+
x,
$end
#300
$dumpon
1!
0"
1#
0$
0%
0&
0'
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1'
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
0!
```

```
1"
0,
0+
0*
0)
0(
0'
0&
0%
0$
#1200
1!
$dumpall
1!
1"
1#
0$
0%
0&
0'
0(
0)
0*
0+
0,
$end
```

Capturing Port Driver Data

Some ASIC vendors' toolkits read a VCD file format that provides details on port drivers. This information can be used, for example, to drive a tester. See the ASIC vendor's documentation for toolkit specific information.

In ModelSim use the vcd dumpports command to create a VCD file that captures port driver data. Each time an external or internal port driver changes values, a new value change is recorded in the VCD file with the following format:

```
p<state> <0 strength> <1 strength> <identifier_code>
```

Driver States

The driver states are recorded as TSSI states if the direction is known, as detailed in this table:

Table 22-4.

Input (testfixture)	Output (dut)
D low	L low
U high	H high
N unknown	X unknown
Z tri-state	T tri-state

Table 22-4.

Input (testfixture)	Output (dut)
d low (two or more drivers active)	l low (two or more drivers active)
u high (two or more drivers active)	h high (two or more drivers active)

If the direction is unknown, the state will be recorded as one of the following:

Table 22-5.

Unknown direction
0 low (both input and output are driving low)
1 high (both input and output are driving high)
? unknown (both input and output are driving unknown)
F three-state (input and output unconnected)
A unknown (input driving low and output driving high)
a unknown (input driving low and output driving unknown)
B unknown (input driving high and output driving low)
b unknown (input driving high and output driving unknown)
C unknown (input driving unknown and output driving low)
c unknown (input driving unknown and output driving high)
f unknown (input and output three-stated)

Driver Strength

The recorded 0 and 1 strength values are based on Verilog strengths:

Table 22-6.

Strength	VHDL std_logic mappings
0 highz	’Z’
1 small	

Table 22-6.

Strength	VHDL std_logic mappings
2 medium	
3 weak	
4 large	
5 pull	’W’,’H’,’L’
6 strong	’U’,’X’,’0’,’1’,’-’
7 supply	

Identifier Code

The <identifier_code> is an integer preceded by < that starts at zero and is incremented for each port in the order the ports are specified. Also, the variable type recorded in the VCD header is "port".

Resolving Values

The resolved values written to the VCD file depend on which options you specify when creating the file.

Default Behavior

By default ModelSim generates output according to IEEE 1364-2005. The standard states that the values 0 (both input and output are active with value 0) and 1 (both input and output are active with value 1) are conflict states. The standard then defines two strength ranges:

- Strong: strengths 7, 6, and 5
- Weak: strengths 4, 3, 2, 1

The rules for resolving values are as follows:

- If the input and output are driving the same value with the same range of strength, the resolved value is 0 or 1, and the strength is the stronger of the two.
- If the input is driving a strong strength and the output is driving a weak strength, the resolved value is D, d, U or u, and the strength is the strength of the input.
- If the input is driving a weak strength and the output is driving a strong strength, the resolved value is L, l, H or h, and the strength is the strength of the output.

Ignoring Strength Ranges

You may wish to ignore strength ranges and have ModelSim handle each strength separately. Any of the following options will produce this behavior:

- Use the **-no_strength_range** argument to the vcd dumpports command
- Use an optional argument to $dumpports (see Extended $dumpports Syntax below)
- Use the **+dumpports+no_strength_range** argument to vsim command

In this situation, ModelSim reports strengths for both the zero and one components of the value if the strengths are the same. If the strengths are different, ModelSim reports only the “winning” strength. In other words, the two strength values either match (e.g., pA 5 5 !) or the winning strength is shown and the other is zero (e.g., pH 0 5 !).

Extended $dumpports Syntax

ModelSim extends the $dumpports system task in order to support exclusion of strength ranges. The extended syntax is as follows:

```
$dumpports (scope_list, file_pathname, ncsim_file_index, file_format)
```

The **nc_sim_index** argument is required yet ignored by ModelSim. It is required only to be compatible with NCSim’s argument list.

The **file_format** argument accepts the following values or an ORed combination thereof (see examples below):

Table 22-7.

File_format value	Meaning
0	Ignore strength range
2	Use strength ranges; produces IEEE 1364-compliant behavior
4	Compress the EVCD output
8	Include port direction information in the EVCD file header; same as using **-direction** argument to **vcd dumpports**

Here are some examples:

```
// ignore strength range
$dumpports(top, "filename", 0, 0)
// compress and ignore strength range
$dumpports(top, "filename", 0, 4)
// print direction and ignore strength range
$dumpports(top, "filename", 0, 8)
```

```
// compress, print direction, and ignore strength range
$dumpports(top, "filename", 0, 12)
```

Example VCD Output from vcd dumpports

This section demonstrates how **vcd dumpports** resolves values based on certain combinations of driver values and strengths and whether or not you use strength ranges.

Sample Driver Data

Table 22-8.

time	in value	out value	in strength value (range)	out strength value (range)
0	0	0	7 (strong)	7 (strong)
100	0	0	6 (strong)	7 (strong)
200	0	0	5 (strong)	7 (strong)
300	0	0	4 (weak)	7 (strong)
900	1	0	6 (strong)	7 (strong)
27400	1	1	5 (strong)	4 (weak)
27500	1	1	4 (weak)	4 (weak)
27600	1	1	3 (weak)	4 (weak)

VCD Output with Strength Ranges

Given the driver data above and use of 1364 strength ranges, here is what the VCD file output would look like:

```
#0
p0 7 0 <0
#100
p0 7 0 <0
#200
p0 7 0 <0
#300
pL 7 0 <0
#900
pB 7 0 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
p1 0 4 <0
```

VCD Output Ignoring Strength Ranges

Here is what the output would look like if you ignore strength ranges:

```
#0
p0 7 0 <0
#100
pL 7 0 <0
#200
pL 7 0 <0
#300
pL 7 0 <0
#900
pL 7 0 <0
#27400
pU 0 5 <0
#27500
p1 0 4 <0
#27600
pH 0 4 <0
```

Chapter 23
Tcl and Macros (DO Files)

Tcl is a scripting language for controlling and extending ModelSim. Within ModelSim you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart ModelSim. In addition, if ModelSim does not provide the command you need, you can use Tcl to create your own commands.

Tcl Features

Using Tcl with ModelSim gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)
- robust scripting language for macros

Tcl References

Two books about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk by* Brent Welch published by Prentice Hall. You can also consult the following online references:

- Select **Help > Tcl Man Pages**.

Tcl Commands

For complete information on Tcl commands, select **Help > Tcl Man Pages**. Also see Simulator GUI Preferences for information on Tcl preference variables.

ModelSim command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

Table 23-1.

Previous ModelSim command	Command changed to (or replaced by)
continue	run with the **-continue** option
format list | wave	write format with either list or wave specified
if	replaced by the Tcl **if** command, see If Command Syntax for more information
list	add list
nolist | nowave	delete with either list or wave specified
set	replaced by the Tcl **set** command, see set Command Syntax for more information
source	vsource
wave	add wave

Tcl Command Syntax

The following eleven rules define the syntax and semantics of the Tcl language. Additional details on If Command Syntax and set Command Syntax follow.

1. A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets ("]") are command terminators during command substitution (see below) unless quoted.
2. A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
3. Words of a command are separated by white space (except for newlines, which are command separators).
4. If the first character of a word is a double-quote (""") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5. If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

6. If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

7. If a word contains a dollar-sign ("$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

   - $name

     Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

   - $name(index)

     Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

   - ${name}

     Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

     There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

8. If a backslash ("\") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without

triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

Table 23-2.

Sequence	**Value**
\a	Audible alert (bell) (0x7)
\b	Backspace (0x8)
\f	Form feed (0xc).
\n	Newline (0xa)
\r	Carriage-return (0xd)
\t	Tab (0x9)
\v	Vertical tab (0xb)
\<newline>whiteSpace	A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.
\\\\	Backslash ("\\")
\ooo	The digits ooo (one, two, or three of them) give the octal value of the character.
\**x***hh*	The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

9. If a hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

10. Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

11. Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

If Command Syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the "?" indicates an optional argument.

Syntax

if *expr1* ?then? *body1* elseif *expr2* ?then? *body2* elseif ... ?else? ?*bodyN*?

Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything else is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

set Command Syntax

The Tcl **set** command reads and writes variables. Note that in the syntax below the "?" indicates an optional argument.

Syntax

set *varName* ?*value*?

Description

Returns the value of variable *varName*. If *value* is specified, then sets the value of *varName* to value, creating a new variable if one doesn't already exist, and returns its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare varName to be a namespace variable.

Command Substitution

Placing a command in square brackets [ ] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

Command Separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

Multiple-Line Commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace '{' is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ"}  {
   echo "Signal value matches"
   do macro_1.do
} else {
   echo "Signal value fails"
   do macro_2.do
}
```

Evaluation Order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else statements, procedures, loops, and so forth.

Tcl Relational Expression Evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

  ```
  if {[exa var_1] == 345}...
  ```

  The following will also work:

  ```
  if {[exa var_1] == "345"}...
  ```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

  ```
  if {[exa var_2] == 001Z}...
  ```

  will give an error.

  ```
  if {[exa var_2] == "001Z"}...
  ```

  will work okay.

- Don't quote single characters in single quotes:

  ```
  if {[exa var_3] == 'X'}...
  ```

  will give an error

  ```
  if {[exa var_3] == "X"}...
  ```

  will work okay.

- For the equal operator, you must use the C operator "==". For not-equal, you must use the C operator "!=".

Variable Substitution

When a $<var_name> is encountered, the Tcl parser will look for variables that have been defined either by ModelSim or by you, and substitute the value of the variable.

Note
Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See Simulator State Variables for more information about ModelSim-defined variables.

System Commands

To pass commands to the UNIX shell or DOS window, use the Tcl **exec** command:

```
echo The date is [exec date]
```

List Processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

Table 23-3.

Command syntax	Description
lappend var_name val1 val2 ...	appends val1, val2, etc. to list var_name
lindex list_name index	returns the index-th element of list_name; the first element is 0
linsert list_name index val1 val2 ...	inserts val1, val2, etc. just before the index-th element of list_name
list val1, val2 ...	returns a Tcl list consisting of val1, val2, etc.
llength list_name	returns the number of elements in list_name
lrange list_name first last	returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list
lreplace list_name first last val1, val2, ...	replaces elements first through last with val1, val2, etc.

Two other commands, **lsearch** and **lsort,** are also available for list manipulation. See the Tcl man pages (**Help** > **Tcl Man Pages**) for more information on these commands.

See also the ModelSim Tcl command: lecho

SimulatorTcl Commands

These additional commands enhance the interface between Tcl and ModelSim. Only brief descriptions are provided here; for more information and command syntax see the Reference Manual.

Table 23-4.

Command	Description
alias	creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias
find	locates incrTcl classes and objects
lecho	takes one or more Tcl lists as arguments and pretty-prints them to the Transcript pane
lshift	takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element
lsublist	returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern
printenv	echoes to the Transcript pane the current names and values of all environment variables

Simulator Tcl Time Commands

ModelSim Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

Conversions

Table 23-5.

Command	Description
intToTime <intHi32> <intLo32>	converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in ModelSim is a 64-bit integer)
RealToTime <real>	converts a <real> number to a 64-bit integer in the current Time Scale
scaleTime <time> <scaleFactor>	returns the value of <time> multiplied by the <scaleFactor> integer

Relations

Table 23-6.

Command	Description
eqTime <time> <time>	evaluates for equal
neqTime <time> <time>	evaluates for not equal
gtTime <time> <time>	evaluates for greater than
gteTime <time> <time>	evaluates for greater than or equal
ltTime <time> <time>	evaluates for less than
lteTime <time> <time>	evaluates for less than or equal

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```
if {[eqTime $Now 1750ns]} {
   ...
}
```

Arithmetic

Table 23-7.

Command	**Description**
addTime <time> <time>	add time
divTime <time> <time>	64-bit integer divide
mulTime <time> <time>	64-bit integer multiply
subTime <time> <time>	subtract time

Tcl Examples

This is an example of using the Tcl **while** loop to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
set i [expr {[llength $a] - 1}]
while {$i >= 0} {
   lappend b [lindex $a $i]
   incr i -1
}
```

This example uses the Tcl **for** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way:

```
set b [list]
for {set i [expr {[llength $a] - 1}]} {$i >= 0} {incr i -1} {
   lappend b [lindex $a $i]
}
```

This example uses the Tcl **foreach** command to copy a list from variable *a* to variable *b*, reversing the order of the elements along the way (the foreach command iterates over all of the elements of a list):

```
set b [list]
foreach i $a { set b [linsert $b 0 $i] }
```

This example shows a list reversal as above, this time aborting on a particular element using the Tcl **break** command:

```
set b [list]
foreach i $a {
   if {$i = "ZZZ"} break
   set b [linsert $b 0 $i]
}
```

This example is a list reversal that skips a particular element by using the Tcl **continue** command:

```
set b [list]
foreach i $a {
   if {$i = "ZZZ"} continue
   set b [linsert $b 0 $i]
}
```

The next example works in UNIX only. In a Windows environment, the Tcl **exec** command will execute compiled files only, not system commands.) The example shows how you can access system information and transfer it into VHDL variables or signals and Verilog nets or registers. When a particular HDL source breakpoint occurs, a Tcl function is called that gets the date and time and deposits it into a VHDL signal of type STRING. If a particular environment variable (DO_ECHO) is set, the function also echoes the new date and time to the transcript file by examining the VHDL variable.

(in VHDL source):

```
signal datime : string(1 to 28) := " ";# 28 spaces
```

(on VSIM command line or in macro):

```
proc set_date {} {
   global env
   set do_the_echo [set env(DO_ECHO)]
   set s [clock format [clock seconds]]
   force -deposit datime $s
   if {do_the_echo} {
      echo "New time is [examine -value datime]"
   }
}

bp src/waveadd.vhd 133 {set_date; continue}
    --sets the breakpoint to call set_date
```

This next example shows a complete Tcl script that restores multiple Wave windows to their state in a previous simulation, including signals listed, geometry, and screen position. It also adds buttons to the Main window toolbar to ease management of the wave files.

```
##  This file contains procedures to manage multiple wave files.
##  Source this file from the command line or as a startup script.
##  source <path>/wave_mgr.tcl
##  add_wave_buttons
##      Add wave management buttons to the main toolbar (new, save and
load)
##  new_wave
##      Dialog box creates a new wave window with the user provided name
##  named_wave <name>
##      Creates a new wave window with the specified title
##  save_wave <file-root>
##      Saves name, window location and contents for all open windows
##  wave windows
##      Creates <file-root><n>.do file for each window where <n> is 1
##      to the number of windows. Default file-root is "wave". Also
##       creates windowSet.do file that contains title and geometry info.
##  load_wave <file-root>
##        Opens and loads wave windows for all files matching <file-
root><n>.do
##      where <n> are the numbers from 1-9. Default <file-root> is "wave".
##       Also runs windowSet.do file if it exists.
## Add wave management buttons to the main toolbar
proc add_wave_buttons {} {
_add_menu main controls right SystemMenu SystemWindowFrame {Load Waves} \
load_wave
_add_menu main controls right SystemMenu SystemWindowFrame {Save Waves} \
save_wave
_add_menu main controls right SystemMenu SystemWindowFrame {New Wave} \
new_wave
}
## Simple Dialog requests name of new wave window. Defaults to Wave<n>

proc new_wave {} {
   global vsimPriv
   set defaultName "Wave[llength $vsimPriv(WaveWindows)]"
   set windowName [GetValue . "Create Named Wave Window:" $defaultName ]
   if {$windowName == ""} {
      # Dialog canceled
      # abort operation
      return
   }
   ## Debug
   puts "Window name: $windowName\n"
   if {$windowName == "{}"} {
      set windowName ""
   }
   if {$windowName != ""} {
      named_wave $windowName
   } else {
      named_wave $defaultName
   }
}

## Creates a new wave window with the provided name (defaults to "Wave")

proc named_wave {{name "Wave"}} {
   set newWave [view -new wave]
   if {[string length $name] > 0} {
      wm title $newWave $name
   }
}

## Writes out format of all wave windows, stores geometry and title info
in
## windowSet.do file. Removes any extra files with the same fileroot.
## Default file name is wave<n> starting from 1.
```

```
proc save_wave {{fileroot "wave"}} {
   global vsimPriv
   set n 1
   if {[catch {open windowSet_$fileroot.do w 755} fileId]} {
      error "Open failure for $fileroot ($fileId)"
   }
   foreach w $vsimPriv(WaveWindows) {
      echo "Saving: [wm title $w]"
      set filename $fileroot$n.do
      if {[file exists $filename]} {
         # Use different file
         set n2 0
         while {[file exists ${fileroot}${n}${n2}.do]} {
            incr n2
         }
         set filename ${fileroot}${n}${n2}.do
      }
      write format wave -window $w $filename
      puts $fileId "wm title $w \"[wm title $w]\""
      puts $fileId "wm geometry $w [wm geometry $w]"
      puts $fileId "mtiGrid_colconfig $w.grid name -width \
         [mtiGrid_colcget $w.grid name -width]"
      puts $fileId "mtiGrid_colconfig $w.grid value -width \
         [mtiGrid_colcget $w.grid value -width]"
      flush $fileId
      incr n
   }

   foreach f [lsort [glob -nocomplain $fileroot\[$n-9\].do]] {
         echo "Removing: $f"
         exec rm $f
      }
   }
}

## Provide file root argument and load_wave restores all saved windows.
## Default file root is "wave".

proc load_wave {{fileroot "wave"}} {
   foreach f [lsort [glob -nocomplain $fileroot\[1-9\].do]] {
      echo "Loading: $f"
      view -new wave
      do $f
   }
   if {[file exists windowSet_$fileroot.do]} {
      do windowSet_$fileroot.do
   }
}

...
```

This next example specifies the compiler arguments and lets you compile any number of files.

```
set Files [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
   set lappend Files $1
   shift
}
eval vcom -93 -explicit -noaccel $Files
```

This example is an enhanced version of the last one. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a *.vhd* file extension.

```
set vhdFiles [list]
set vFiles [list]
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
   if {[string match *.vhd $1]} {
      lappend vhdFiles $1
   } else {
      lappend vFiles $1
   }
   shift
}
if {[llength $vhdFiles] > 0} {
   eval vcom -93 -explicit -noaccel $vhdFiles
}
if {[llength $vFiles] > 0} {
   eval vlog $vFiles
}
```

Macros (DO Files)

ModelSim macros (also called DO files) are simply scripts that contain ModelSim and, optionally, Tcl commands. You invoke these scripts with the **Tools** > **Execute Macro** menu selection or the do command.

Creating DO Files

You can create DO files, like any other Tcl script, by typing the required commands in any editor and saving the file. Alternatively, you can save the transcript as a DO file (see Saving the Transcript File).

All "event watching" commands (e.g. onbreak, onerror, etc.) must be placed before run commands within the macros in order to take effect.

The following is a simple DO file that was saved from the transcript. It is used in the dataset exercise in the ModelSim Tutorial. This DO file adds several signals to the Wave window, provides stimulus to those signals, and then advances the simulation.

```
add wave ld
add wave rst
add wave clk
add wave d
add wave q
force -freeze clk 0 0, 1 {50 ns} -r 100
force rst 1
force rst 0 10
force ld 0
force d 1010
onerror {cont}
run 1700
force ld 1
run 100
force ld 0
run 400
force rst 1
run 200
force rst 0 10
run 1500
```

Using Parameters with DO Files

You can increase the flexibility of DO files by using parameters. Parameters specify values that are passed to the corresponding parameters $1 through $9 in the macro file. For example say the macro "*testfile"* contains the line **bp** $1 $2. The command below would place a breakpoint in the source file named *design.vhd* at line 127:

do testfile design.vhd 127

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the shift command to see the other parameters.

Deleting a File from a .do Script

To delete a file from a .do script, use the Tcl **file** command as follows:

file delete myfile.log

This will delete the file "*myfile.log*."

You can also use the **transcript file** command to perform a deletion:

transcript file ()
transcript file my file.log

The first line will close the current log file. The second will open a new log file. If it has the same name as an existing file, it will replace the previous one.

Making Macro Parameters Optional

If you want to make macro parameters optional (i.e., be able to specify fewer parameter values with the do command than the number of parameters referenced in the macro), you must use the **argc** simulator state variable. The **argc** simulator state variable returns the number of parameters passed. The examples below show several ways of using **argc**.

Example 1

This macro specifies the files to compile and handles 0-2 compiler arguments as parameters. If you supply more arguments, ModelSim generates a message.

```
switch $argc {
  0 {vcom file1.vhd file2.vhd file3.vhd }
  1 {vcom $1 file1.vhd file2.vhd file3.vhd }
  2 {vcom $1 $2 file1.vhd file2.vhd file3.vhd }
  default {echo Too many arguments. The macro accepts 0-2 args.  }
}
```

Example 2

This macro specifies the compiler arguments and lets you compile any number of files.

```
variable Files ""
set nbrArgs $argc
for {set x 1} {$x <= $nbrArgs} {incr x} {
  set Files [concat $Files $1]
  shift
}
eval vcom -93 -explicit -noaccel $Files
```

Example 3

This macro is an enhanced version of the one shown in example 2. The additional code determines whether the files are VHDL or Verilog and uses the appropriate compiler and arguments depending on the file type. Note that the macro assumes your VHDL files have a .vhd file extension.

```
variable vhdFiles ""
variable vFiles ""
set nbrArgs $argc
set vhdFilesExist 0
set vFilesExist   0
for {set x 1} {$x <= $nbrArgs} {incr x} {
  if {[string match *.vhd $1]} {
    set vhdFiles [concat $vhdFiles $1]
    set vhdFilesExist 1
  } else {
    set vFiles [concat $vFiles $1]
    set vFilesExist 1
  }
  shift
}
if {$vhdFilesExist == 1} {
  eval vcom -93 -explicit -noaccel $vhdFiles
}
if {$vFilesExist == 1} {
  eval vlog $vFiles
}
```

Useful Commands for Handling Breakpoints and Errors

If you are executing a macro when your simulation hits a breakpoint or causes a run-time error, ModelSim interrupts the macro and returns control to the command line. The following commands may be useful for handling such events. (Any other legal command may be executed as well.)

Table 23-8.

command	**result**
run **-continue**	continue as if the breakpoint had not been executed, completes the run that was interrupted
resume	continue running the macro
onbreak	specify a command to run when you hit a breakpoint within a macro
onElabError	specify a command to run when an error is encountered during elaboration
onerror	specify a command to run when an error is encountered within a macro
status	get a traceback of nested macro calls when a macro is interrupted
abort	terminate a macro once the macro has been interrupted or paused

Table 23-8.

command	result
pause	cause the macro to be interrupted; the macro can be resumed by entering a resume command via the command line
transcript	control echoing of macro commands to the Transcript pane

You can also set the OnErrorDefaultAction Tcl variable to determine what action ModelSim takes when an error occurs. To set the variable on a permanent basis, you must define the variable in a *modelsim.tcl* file (see The modelsim.tcl File for details).

Error Action in DO Files

If a command in a macro returns an error, ModelSim does the following:

1. If an onerror command has been set in the macro script, ModelSim executes that command. The onerror command must be placed prior to the run command in the DO file to take effect.
2. If no **onerror** command has been specified in the script, ModelSim checks the OnErrorDefaultAction variable. If the variable is defined, its action will be invoked.
3. If neither 1 or 2 is true, the macro aborts.

Using the Tcl Source Command with DO Files

Either the **do** command or Tcl **source** command can execute a DO file, but they behave differently.

With the **source** command, the DO file is executed exactly as if the commands in it were typed in by hand at the prompt. Each time a breakpoint is hit, the Source window is updated to show the breakpoint. This behavior could be inconvenient with a large DO file containing many breakpoints.

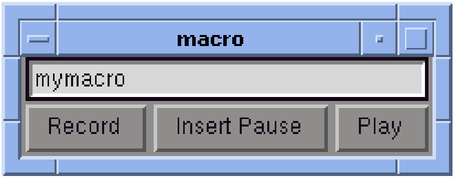
When a **do** command is interrupted by an error or breakpoint, it does not update any windows, and keeps the DO file "locked". This keeps the Source window from flashing, scrolling, and moving the arrow when a complex DO file is executed. Typically an **onbreak resume** command is used to keep the macro running as it hits breakpoints. Add an **onbreak abort** command to the DO file if you want to exit the macro and update the Source window.

Macro Helper

This tool is available for UNIX only (excluding Linux).

The purpose of the Macro Helper is to aid macro creation by recording a simple series of mouse movements and key strokes. The resulting file can be called from a more complex macro by using the play command. Actions recorded by the Macro Helper can only take place within the ModelSim GUI (window sizing and repositioning are not recorded because they are handled by your operating system's window manager). In addition, the run commands cannot be recorded with the Macro Helper but can be invoked as part of a complex macro.

Select **Tools** > **Macro Helper** to access the Macro Helper.



- **Record a macro**
  by typing a new macro file name into the field provided and pressing **Record**.
- **Play a macro**
  by entering the file name of a Macro Helper file into the field and pressing **Play**.

Files created by the Macro Helper can be viewed with the notepad command.

See the macro_option command for playback speed, delay, and debugging options for completed macro files.

The Tcl Debugger

We would like to thank Gregor Schmid for making TDebug available for use in the public domain.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of FITNESS FOR A PARTICULAR PURPOSE.

Starting the Debugger

Select **Tools** > **Tcl Debugger** to run the debugger. Make sure you use the ModelSim and TDebug menu selections to invoke and close the debugger. If you would like more information on the configuration of TDebug see **Help** > **Technotes** > **tdebug**.

The following text is an edited summary of the README file distributed with TDebug.

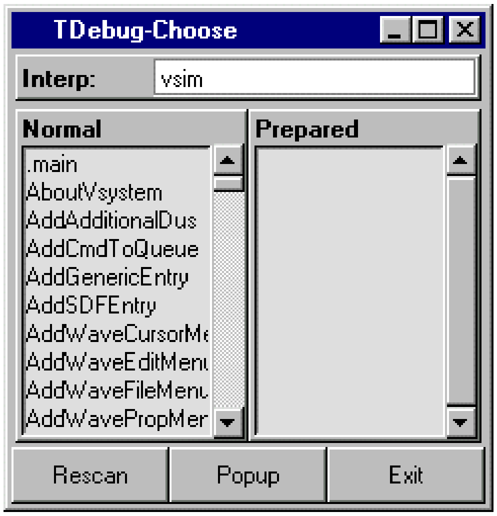
How it Works

TDebug works by parsing and redefining Tcl/Tk-procedures, inserting calls to `td_eval' at certain points, which takes care of the display, stepping, breakpoints, variables etc. The advantages are that TDebug knows which statement in which procedure is currently being executed and can give visual feedback by highlighting it. All currently accessible variables and their values are displayed as well. Code can be evaluated in the context of the current procedure. Breakpoints can be set and deleted with the mouse.

Unfortunately there are drawbacks to this approach. Preparation of large procedures is slow and due to Tcl's dynamic nature there is no guarantee that a procedure can be prepared at all. This problem has been alleviated somewhat with the introduction of partial preparation of procedures. There is still no possibility to get at code running in the global context.

The Chooser

Select **Tools** > **Tcl Debugger** to open the TDebug chooser.



Interp: vsim

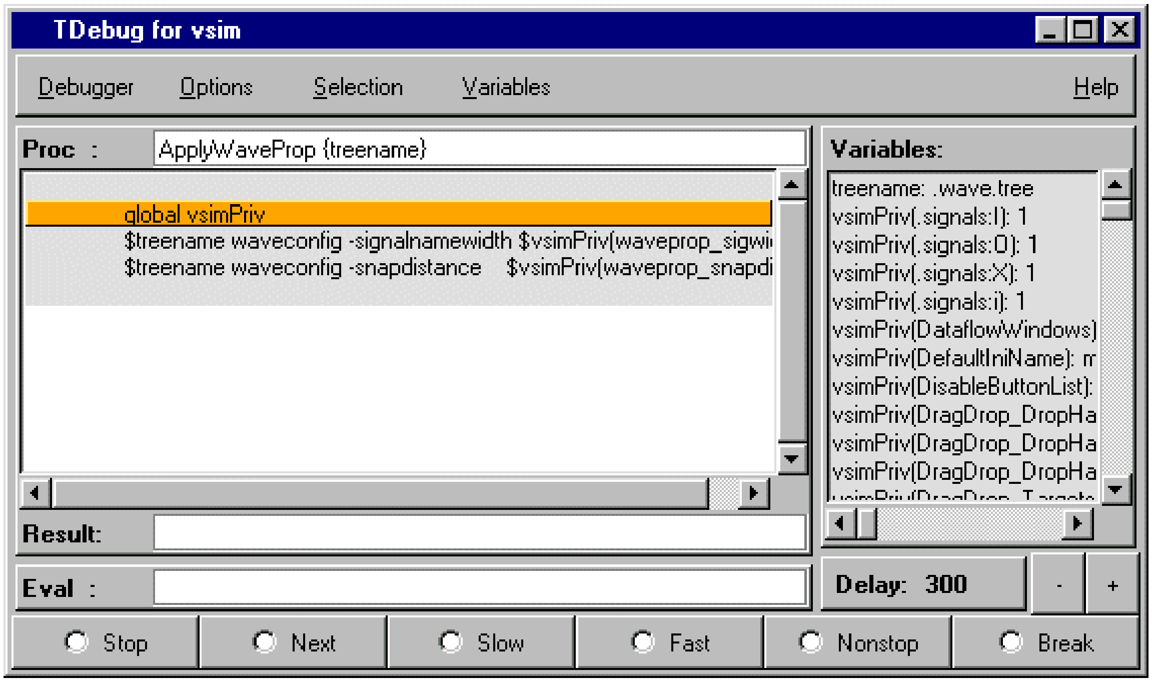
The TDebug chooser has three parts. At the top the current interpreter, *vsim.op_*, is shown. In the main section there are two list boxes. All currently defined procedures are shown in the left list box. By clicking the left mouse button on a procedure name, the procedure gets prepared for debugging and its name is moved to the right list box. Clicking a name in the right list box returns a procedure to its normal state.

Press the right mouse button on a procedure in either list box to get its program code displayed in the main debugger window.

The three buttons at the bottom let you force a **Rescan** of the available procedures, **Popup** the debugger window or **Exit** TDebug. Exiting from TDebug doesn't terminate ModelSim, it merely detaches from *vsim.op_*, restoring all prepared procedures to their unmodified state.

The Debugger

Select the **Popup** button in the Chooser to open the debugger window.



Proc : ApplyWaveProp {treename}

```
        global vsimPriv
        $treename waveconfig -signalnamewidth $vsimPriv(waveprop_sigwi
        $treename waveconfig -snapdistance    $vsimPriv(waveprop_snapdi
```

Variables:

```
treename: .wave.tree
vsimPriv(.signals:I): 1
vsimPriv(.signals:O): 1
vsimPriv(.signals:X): 1
vsimPriv(.signals:i): 1
vsimPriv(DataflowWindows)
vsimPriv(DefaultIniName): m
vsimPriv(DisableButtonList):
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
```

Result:

Eval :

Delay: 300

The debugger window is divided into the main region with the name of the current procedure (**Proc**), a listing in which the expression just executed is highlighted, the **Result** of this execution and the currently available **Variables** and their values, an entry to **Eval** expressions in the context of the current procedure, and some button controls for the state of the debugger.

A procedure listing displayed in the main region will have a darker background on all lines that have been prepared. You can prepare or restore additional lines by selecting a region (<Button-1>, standard selection) and choosing **Selection > Prepare Proc** or **Selection > Restore Proc** from the debugger menu (or by pressing ^P or ^R).

When using `Prepare' and `Restore', try to be smart about what you intend to do. If you select just a single word (plus some optional white space) it will be interpreted as the name of a procedure to prepare or restore. Otherwise, if the selection is owned by the listing, the corresponding lines will be used.

Be careful with partial prepare or restore! If you prepare random lines inside a `switch' or `bind' expression, you may get surprising results on execution, because the parser doesn't know about the surrounding expression and can't try to prevent problems.

There are seven possible debugger states, one for each button and an `idle' or `waiting' state when no button is active. The button-activated states are:

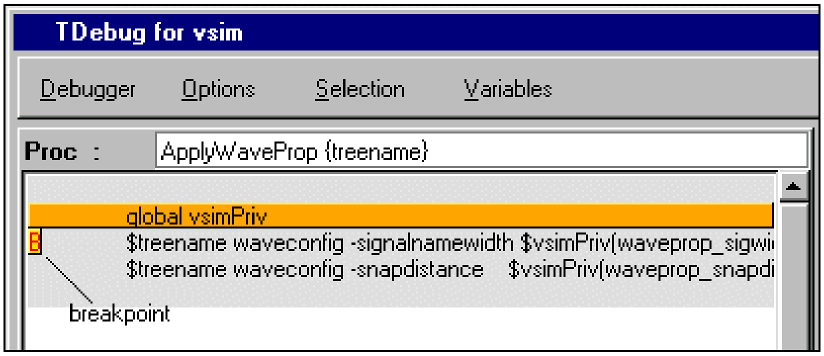
Table 23-9.

Button	Description
Stop	stop after next expression, used to get out of slow/fast/nonstop mode
Next	execute one expression, then revert to idle
Slow	execute until end of procedure, stopping at breakpoints or when the state changes to stop; after each execution, stop for ’delay’ milliseconds; the delay can be changed with the ’+’ and ’-’ buttons
Fast	execute until end of procedure, stopping at breakpoints
Nonstop	execute until end of procedure without stopping at breakpoints or updating the display
Break	terminate execution of current procedure

Closing the debugger doesn't quit it, it only does `wm withdraw'. The debugger window will pop up the next time a prepared procedure is called. Make sure you close the debugger with **Debugger > Close**.

Breakpoints

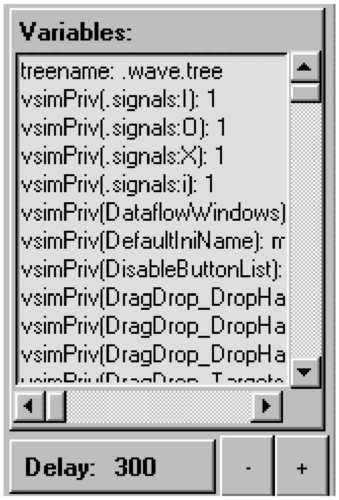
To set/unset a breakpoint, double-click inside the listing. The breakpoint will be set at the innermost available expression that contains the position of the click. Conditional or counted breakpoints aren’t supported.



```
global vsimPriv
$treename waveconfig -signalnamewidth $vsimPriv(waveprop_sigwi
$treename waveconfig -snapdistance    $vsimPriv(waveprop_snapdi
```

The **Eval** entry supports a simple history mechanism available via the <Up_arrow> and <Down_arrow> keys. If you evaluate a command while stepping through a procedure, the command will be evaluated in the context of the procedure; otherwise it will be evaluated at the global level. The result will be displayed in the result field. This entry is useful for a lot of things, but especially to get access to variables outside the current scope.

Try entering the line `global td_priv' and watch the **Variables** box (with global and array variables enabled of course).



```
treename: .wave.tree
vsimPriv(.signals:I): 1
vsimPriv(.signals:O): 1
vsimPriv(.signals:X): 1
vsimPriv(.signals:i): 1
vsimPriv(DataflowWindows)
vsimPriv(DefaultIniName): m
vsimPriv(DisableButtonList):
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
vsimPriv(DragDrop_DropHa
```

Configuration

You can customize TDebug by setting up a file named *.tdebugrc* in your home directory. See the TDebug README at **Help** > **Technotes** > **tdebug** for more information on the configuration of TDebug.

TclPro Debugger

The Tools menu in the Main window contains a selection for the TclPro Debugger from Scriptics Corporation. This debugger and any available documentation can be acquired from Scriptics. Once acquired, do the following steps to use the TclPro Debugger:

1. Make sure the TclPro bin directory is in your PATH.
2. In TclPro Debugger, create a new project with Remote Debugging enabled.
3. Start ModelSim and select **Tools** > **TclPro Debugger**.
4. Press the Stop button in the debugger in order to set breakpoints, etc.

Note

TclPro Debugger version 1.4 does not work with ModelSim.

Appendix A
Simulator Variables

This appendix documents the following types of variables:

- **environment variables** — Variables referenced and set according to operating system conventions. Environment variables prepare the ModelSim environment prior to simulation.
- **simulator control variables —** Variables used to control compiler, simulator , and various other functions.
- **simulator state variable**s — Variables that provide feedback on the state of the current simulation.

Variable Settings Report

The report command returns a list of current settings for either the simulator state or simulator control variables. Use the following commands at either the ModelSim or VSIM prompt:

```
report simulator state
report simulator control
```

The simulator control variables reported by the **report simulator control** command can be set interactively using the Tcl set Command Syntax.

Environment Variables

Environment Variable Expansion

The shell commands vcom, vlog, vsim, and vmap, no longer expand environment variables in filename arguments and options. Instead, variables should be expanded by the shell beforehand, in the usual manner. The -f option that most of these commands support, now performs environment variable expansion throughout the file.

Environment variable expansion is still performed in the following places:

- Pathname and other values in the *modelsim.ini* file
- Strings used as file pathnames in VHDL and Verilog
- VHDL Foreign attributes

- The PLIOBJS environment variable may contain a path that has an environment variable.
- Verilog `uselib file and dir directives
- Anywhere in the contents of a -f file

The recommended method for using flexible pathnames is to make use of the MGC Location Map system (see Using Location Mapping). When this is used, then pathnames stored in libraries and project files (.mpf) will be converted to logical pathnames.

If a file or path name contains the dollar sign character ($), and must be used in one of the places listed above that accepts environment variables, then the explicit dollar sign must be escaped by using a double dollar sign ($$).

Setting Environment Variables

Before compiling or simulating, several environment variables may be set to provide the functions described below. The variables are set through the System control panel on Windows 2000 and XP machines. For UNIX, the variables are typically found in the *.login* script. The LM_LICENSE_FILE variable is required; all others are optional.

DOPATH

The toolset uses the DOPATH environment variable to search for DO files (macros). DOPATH consists of a colon-separated (semi-colon for Windows) list of paths to directories. You can override this environment variable with the DOPATH Tcl preference variable.

The DOPATH environment variable isn’t accessible when you invoke vsim from a Unix shell or from a Windows command prompt. It is accessible once QuestaSim or vsim is invoked. If you need to invoke from a shell or command line and use the DOPATH environment variable, use the following syntax:

```
vsim -do "do <dofile_name>" <design_unit>
```

EDITOR

The EDITOR environment variable specifies the editor to invoke with the edit command

HOME

The toolset uses the HOME environment variable to look for an optional graphical preference file and optional location map file. Refer to Control Variables Located in INI Files for additional information.

HOME_0IN

The HOME_0IN environment variable identifies the location of the 0-In executables directory. Refer to the 0-In documentation for more information

LM_LICENSE_FILE

The toolset's file manager uses the LM_LICENSE_FILE environment variable to find the location of the license file. The argument may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files. The environment variable is required.

MODEL_TECH

The toolset automatically sets the MODEL_TECH environment variable to the directory in which the binary executable resides; DO NOT SET THIS VARIABLE!

MODEL_TECH_TCL

The toolset uses the MODEL_TECH_TCL environment variable to find Tcl libraries for Tcl/Tk 8.3 and vsim, and may also be used to specify a startup DO file. This variable defaults to */modeltech/../tcl*, however you may set it to an alternate path

MGC_LOCATION_MAP

The toolset uses the MGC_LOCATION_MAP environment variable to find source files based on easily reallocated "soft" paths.

MODELSIM

The toolset uses the MODELSIM environment variable to find the *modelsim.ini* file. The argument consists of a path including the file name.

An alternative use of this variable is to set it to the path of a project file (*<Project_Root_Dir>/<Project_Name>.mpf*). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace *modelsim.ini* as the initialization file for all tools.

MODELSIM_PREFERENCES

The MODELSIM_PREFERENCES environment variable specifies the location to store user interface preferences. Setting this variable with the path of a file instructs the toolset to use this file instead of the default location (your HOME directory in UNIX or in the registry in Windows). The file does not need to exist beforehand, the toolset will initialize it. Also, if this file is read-only, the toolset will not update or otherwise modify the file. This variable may contain a relative pathname – in which case the file will be relative to the working directory at the time the tool is started.

MODELSIM_TCL

The toolset uses the MODELSIM_TCL environment variable to look for an optional graphical preference file. The argument can be a colon-separated (UNIX) or semi-colon separated (Windows) list of file paths.

MTI_COSIM_TRACE

The MTI_COSIM_TRACE environment variable creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls. You should set this variable to any value before invoking the simulator.

MTI_TF_LIMIT

The MTI_TF_LIMIT environment variable limits the size of the VSOUT temp file (generated by the toolset's kernel). Set the argument of this variable to the size of k-bytes

The environment variable TMPDIR controls the location of this file, while STDOUT controls the name. The default setting is 10, and a value of 0 specifies that there is no limit. This variable does *not* control the size of the transcript file.

MTI_RELEASE_ON_SUSPEND

The MTI_RELEASE_ON_SUSPEND environment variable allows you to turn off or modify the delay for the functionality of releasing all licenses when the tool is suspended. The default setting is 10 (in seconds), which means that if you do not set this variable your licenses will be released 10 seconds after your run is suspended. If you set this environment variable with an argument of 0 (zero) the tool will not release the licenses after being suspended. You can change the default length of time (number of seconds) by setting this environment variable to an integer greater than 0 (zero).

MTI_USELIB_DIR

The MTI_USELIB_DIR environment variable specifies the directory into which object libraries are compiled when using the **-compile_uselibs** argument to the vlog command

MTI_VCO_MODE

The MTI_VCO_MODE environment variable determines which version of the toolset to use on platforms that support both 32- and 64-bit versions when the executables are invoked from the *modeltech/bin* directory by a Unix shell command (using full path specification or PATH search_). Acceptable values are either "32" or "64" (do not include quotes). If you do not set this variable, the preference is given to the highest performance installed version.

NOMMAP

When set to 1, the NOMMAP environment variable disables memory mapping in the toolset. You should only use this variable when running on Linux 7.1 because it will decrease the speed with which the tool reads files.

PLIOBJS

The toolset uses the PLIOBJS environment variable to search for PLI object files for loading. The argument consists of a space-separated list of file or path names

STDOUT

The argument to the STDOUT environment variable specifies a filename to which the simulator saves the VSOUT temp file information. Typically this information is deleted when the simulator exits. The location for this file is set with the TMPDIR variable, which allows you to find and delete the file in the event of a crash, because an unnamed VSOUT file is not deleted after a crash.

TMP

(Windows environments) The TMP environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

TMPDIR

(UNIX environments) The TMPDIR environment variable specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel.

Creating Environment Variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

Using Windows 2000 and XP

Right-click the **My Computer** icon and select **Properties**, then select the Advanced tab and then Environment Variables. Add the new variable with this data—Variable:*MY_PATH* and Value:*\temp\work.*

Click **Set** and **Apply** to initialize the variable.

Library Mapping with Environment Variables

Once the **MY_PATH** variable is set, you can use it with the vmap command to add library mappings to the current *modelsim.ini* file.

If you’re using the **vmap** command from a DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you’re using **vmap** from the ModelSim/VSIM prompt type:

```
vmap MY_VITAL \$MY_PATH
```

If you used DOS **vmap**, this line will be added to the *modelsim.ini*:

```
MY_VITAL = c:\temp\work
```

If **vmap** is used from the ModelSim/VSIM prompt, the *modelsim.ini* file will be modified with this line:

```
MY_VITAL = $MY_PATH
```

You can easily add additional hierarchy to the path. For example,

```
vmap MORE_VITAL %MY_PATH%\more_path\and_more_path
```

```
vmap MORE_VITAL \$MY_PATH\more_path\and_more_path
```

The "$" character in the examples above is Tcl syntax that precedes a variable. The "\" character is an escape character that keeps the variable from being evaluated during the execution of **vmap**.

Referencing Environment Variables

There are two ways to reference environment variables within ModelSim. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
use std.textio.all;
entity test is end;
architecture only of test is
begin
   process
      FILE in_file : text is in "$ENV_VAR_NAME";
   begin
      wait;
   end process;
end;
```

Environment variables may also be referenced from the ModelSim command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

Note

Environment variable expansion *does not* occur in files that are referenced via the **-f** argument to **vcom**, **vlog**, or **vsim**.

Removing Temp Files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the ModelSim GUI. In normal circumstances the file is deleted when the simulator exits. If ModelSim crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

Control Variables Located in INI Files

Initialization (INI) files contain control variables that specify reference library paths and compiler and simulator settings. The default initialization file is *modelsim.ini* and is located in your install directory.

To set these variables, edit the initialization file directly with any text editor. The syntax for variables in the file is:

<variable> = <value>

Comments within the file are preceded with a semicolon (;).

Table A-1.

INI file sections
Library Path Variables
Verilog Compiler Control Variables
VHDL Compiler Control Variables
SystemC Compiler Control Variables
Simulator Control Variables
Logic Modeling Variables

Library Path Variables

You can find these variables under the heading [Library].

ieee

This variable sets the path to the library containing IEEE and Synopsys arithmetic packages.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../ieee

modelsim_lib

This variable sets the path to the library containing Model Technology VHDL utilities such as Signal Spy.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../modelsim_lib

std

This variable sets the path to the VHDL STD library.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../std

std_developerskit

This variable sets the path to the libraries for MGC standard developer's kit.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../std_developerskit

synopsys

This variable sets the path to the accelerated arithmetic packages.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../synopsys

sv_std

This variable sets the path to the SystemVerilog STD library.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../sv_std

verilog

This variable sets the path to the library containing VHDL/Verilog type mappings.

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../verilog

vital2000

This variable sets the path to the VITAL 2000 library

- **Value Range**: any valid path; may include environment variables
- **Default**: $MODEL_TECH/../vital2000

others

This variable points to another *modelsim.ini* file whose library path variables will also be read; the pathname must include "modelsim.ini"; only one others variable can be specified in any *modelsim.ini* file.

- **Value Range**: any valid path; may include environment variables
- **Default**: none

Verilog Compiler Control Variables

You can find these variables under the heading [vlog].

DisableOpt

This variable, when on, disables all optimizations enacted by the compiler; same as the **-O0** argument to **vlog**.

- **Value Range**: 0, 1
- **Default**: off (0)

Hazard

This variable turns on Verilog hazard checking (order-dependent accessing of global variables).

- **Value Range**: 0, 1
- **Default**: off (0)

GenerateLoopIterationMax

This variable specifies the maximum number of iterations permitted for a generate loop; restricting this permits the implementation to recognize infinite generate loops.

- **Value Range**: natural integer (>=0)
- **Default**: 100000

GenerateRecursionDepthMax

This variable specifies the maximum depth permitted for a recursive generate instantiation; restricting this permits the implementation to recognize infinite recursions.

- **Value Range**: natural integer (>=0)
- **Default**: 200

Incremental

This variable activates the incremental compilation of modules.

- **Value Range**: 0, 1
- **Default**: off (0)

MultiFileCompilationUnit

Controls how Verilog files are compiled into compilation units. Valid arguments:

- 1 -- (0n) Compiles all files on command line into a single compilation unit. This behavior is called Multi File Compilation Unit (MFCU) mode; same as -mfcu argument to
- 0 -- (Off) Default value. Compiles each file in the compilation command line into separate compilation units. This behavior is called Single File Compilation Unit (SFCU) mode.

Refer to SystemVerilog Multi-File Compilation Issues for details on the implications of these settings. Refer to

Note

The default behavior in versions prior to 6.1 was opposite of the current default behavior.

NoDebug

This variable, when on, disables the inclusion of debugging info within design units.

- **Value Range**: 0, 1
- **Default**: off (0)

Protect

This variable enables `protect directive processing. Refer to Compiler Directives for details.

- **Value Range**: 0, 1
- **Default**: off (0)

Quiet

This variable turns off "loading…" messages.

- **Value Range**: 0, 1
- **Default**: off (0)

ScalarOpts

This variable activates optimizations on expressions that do not involve signals, waits, or function/procedure/task invocations.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_BadOptionWarning

This variable instructs the tool to generate a warning whenever an unknown plus argument is encountered.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_Lint

This variable instructs the tool to display lint warning messages.

- **Value Range**: 0, 1
- **Default**: off (0)

Show_WarnCantDoCoverage

This variable instructs the tool to display warning messages when the simulator encounters constructs which code coverage cannot handle.

- **Value Range**: 0,1
- **Default**: on (1)

Show_WarnMatchCadence

This variable instructs the tool to display warning messages about non-LRM compliance in order to match Cadence behavior.

- **Value Range**: 0, 1
- **Default**: on (1)

Show_source

This variable instructs the tool to show any source line containing an error.

- **Value Range**: 0, 1
- **Default**: off (0)

SparseMemThreshhold

This variable specifies the size at which memories will automatically be marked as sparse memory. Refer to Sparse Memory Modeling for more information.

- **Value Range**: natural integer (>=0)

- **Default**: off (0)

UpCase

This variable instructs the tool to activate the conversion of regular Verilog identifiers to uppercase and allows case insensitivity for module names. Refer to Verilog-XL Compatible Compiler Arguments for more information.

- **Value Range**: 0, 1
- **Default**: off (0)

vlog95compat

This variable instructs the tool to disable SystemVerilog and Verilog 2001 support, making the compiler compatible with IEEE Std 1364-1995.

- **Value Range**: 0, 1
- **Default**: off (0)

VlogZeroIn

This variable instructs **vlog** to automatically invoke **0in analyze**; same as **vlog -0in**

- **Value Range**: 0, 1
- **Default**: off (0)

VlogZeroInOptions

This variable passes options to **0in ccl**.

- **Value Range**: any valid 0-In options
- **Default**: null

VoptZeroIn

This variable instructs **vopt** to automatically invoke **0in ccl**; same as **vopt -0in**

- **Value Range**: 0, 1
- **Default**: off (0)

VoptZeroInOptions

This variable passes options to **0in ccl**.

- **Value Range**: any valid 0-In options

- **Default**: null

VHDL Compiler Control Variables

You can find these variables under the heading [vcom].

BindAtCompile

This variable instructs the tool to perform VHDL default binding at compile time rather than load time. Refer to Default Binding for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

CheckSynthesis

This variable turns on limited synthesis rule compliance checking, which includes checking only signals used (read) by a process and understanding only combinational logic, not clocked logic.

- **Value Range:** 0, 1
- **Default:** off (0)

DisableOpt

This variable disables all optimizations enacted by the compiler, similar to using the **-O0** argument to **vcom**.

- **Value Range:** 0, 1
- **Default:** off (0)

Explicit

This variable enables the resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration).

- **Value Range:** 0, 1
- **Default:** on (1)

IgnoreVitalErrors

This variable instructs the tool to ignore VITAL compliance checking errors.

- **Value Range:** 0, 1

- **Default:** off (0)

NoCaseStaticError

This variable changes case statement static errors to warnings.

- **Value Range:** 0, 1
- **Default:** off (0)

NoDebug

This variable disables turns off inclusion of debugging info within design units.

- **Value Range:** 0, 1
- **Default:** off (0)

NoIndexCheck

This variable disables run time index checks.

- **Value Range:** 0, 1
- **Default:** off (0)

NoOthersStaticError

This variable disables errors caused by aggregates that are not locally static.

- **Value Range:** 0, 1
- **Default:** off (0)

NoRangeCheck

This variable disables run time range checking.

- **Value Range:** 0, 1
- **Default:** off (0)

NoVital

This variable disables acceleration of the VITAL packages.

- **Value Range:** 0, 1
- **Default:** off (0)

NoVitalCheck

This variable disables VITAL compliance checking.

- **Value Range:** 0, 1
- **Default:** off (0)

Optimize_1164

This variable disables optimization for the IEEE std_logic_1164 package.

- **Value Range:** 0, 1
- **Default:** on (1)

PedanticErrors

This variable overrides NoCaseStaticError and NoOthersStaticError

- **Value Range:** 0, 1
- **Default:** off(0)

Quiet

This variable disables the “loading…” messages.

- **Value Range:** 0, 1
- **Default:** off (0)

RequireConfigForAllDefaultBinding

This variable instructs the compiler not to generate a default binding during compilation.

- **Value Range:** 0, 1
- **Default:** off (0)

ScalarOpts

This variable activates optimizations on expressions that do not involve signals, waits, or function/procedure/task invocations.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_Lint

This variable enables lint-style checking.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_source

This variable shows source line containing error.

- **Value Range:** 0, 1
- **Default:** off (0)

Show_VitalChecksOpt

This variable enables VITAL optimization warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_VitalChecksWarnings

This variable enables VITAL compliance-check warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_WarnCantDoCoverage

This variable enables warnings when the simulator encounters constructs which code coverage cannot handle.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning1

This variable enables unbound-component warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning2

This variable enables process-without-a-wait-statement warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning3

This variable enables null-range warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning4

This variable enables no-space-in-time-literal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning5

This variable enables multiple-drivers-on-unresolved-signal warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning9

This variable enables warnings about signal value dependency at elaboration.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_Warning10

This variable enables warnings about VHDL-1993 constructs in VHDL-1987 code.

- **Value Range:** 0, 1
- **Default:** on (1)

Show_WarnLocallyStaticError

This variable enables warnings about locally static errors deferred until run time.

- **Value Range:** 0, 1
- **Default:** on (1)

VHDL93

This variable enables support for VHDL-1987, where “1” enables support for VHDL-1993 and “2” enables support for VHDL-2002.

- **Value Range:** 0, 1, 2
- **Default:** 2

SystemC Compiler Control Variables

You can find these variables under the heading [sccom].

CppOptions

This variable adds any specified C++ compiler options to the **sccom** command line at the time of invocation.

- **Value Range:** any valid C+++ compiler options
- **Default:** none

CppPath

This variable should point directly to the location of the g++ executable, such as:

```
% CppPath /usr/bin/g++
```

This variable is not required when running SystemC designs. By default, you should install and use the built-in g++ compiler that comes with the tool.

- **Value Range:** C++ compiler path
- **Default:** none

DpiOutOfTheBlue

This variable enables DPI “out of the blue” calls from C functions (must not be declared as import tasks or functions).

- **Value Range:** 0, 1
- **Default:** 0 - Support for “out of the blue” DPI calls is disabled.

NoExitOnScStop

This variable controls whether the tool exits when it encounters an sc_stop() in the design. When enabled, control is returned to the VSIM> prompt upon execution of sc_stop().

- **Value Range:** 0, 1
- **Default:** off (0) - The tool exits when it encounters sc_stop in code.

RetroChannelLimit

This variable controls the maximum number of retroactive recording channels allowed in the WLF file. Setting the value to 0 turns off retroactive recording. Setting the value too high can risk your performance, and the WLF file may not operate.

- **Value Range:** integer
- **Default:** 20

SccomLogfile

This variable creates a log file for sccom.

- **Value Range:** 0, 1
- **Default:** off (0)

SccomVerbose

This variable enables verbose messages from sccom, refer to sccom -verbose for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

ScvPhaseRelationName

This variable changes the precise name used by SCV to specify “phase” transactions in the WLF file.

- **Value Range:** any legal string is accepted, but legal C-language identifiers are recommended.
- **Default:** mti_phase

UseScv

This variable enables the use of SCV include files and library, refer to sccom -scv for more information.

- **Value Range:** 0, 1
- **Default:** off (0)

Simulator Control Variables

You can find these variables under the heading [vsim].

AssertFile

This variable specifies an alternative file for storing VHDL assertion messages.

- **Value Range:** any valid filename
- **Default:** transcript

AssertionDebug

This variable specifies that SVA assertion passes are reported.

- **Value Range:** 0, 1
- **Default:** off (0)

AssertionFormat

This variable defines the format of VHDL assertion messages.

- **Value Range:**

Table A-2.

Variable	Description
%S	severity level
%R	report message
%T	time of assertion
%D	delta
%I	instance or region pathname (if available)
%i	instance pathname with process
%O	process name
%K	kind of object path points to; returns Instance, Signal, Process, or Unknown
%P	instance or region path without leaf process
%F	file

Table A-2.

Variable	Description
%L	line number of assertion, or if from subprogram, line from which call is made
%%	print ’%’ character

- Default: "** %S: %R\n Time: %T Iteration: %D%I\n"

AssertionFormatBreak

This variable defines the format of messages for VHDL assertions that trigger a breakpoint.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatError

This variable defines the format of messages for VHDL Error assertions.

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatFail

This variable defines the format of messages for VHDL Fail assertions.

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatFatal

This variable defines the format of messages for VHDL Fatal assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used.

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D %K: %i File: %F\n"

AssertionFormatNote

This variable defines the format of messages for VHDL Note assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D%I\n"

AssertionFormatWarning

This variable defines the format of messages for VHDL Warning assertions

If undefined, AssertionFormat is used unless assertion causes a breakpoint in which case AssertionFormatBreak is used

- **Value Range:** Refer to Table A-2
- **Default:** "** %S: %R\n   Time: %T Iteration: %D%I\n"

BreakOnAssertion

This variable defines the severity of VHDL assertions that cause a simulation break. It also controls any messages in the source code that use *assertion_failure_**. For example, since most runtime messages use some form of assertion_failure_*, any runtime error will cause the simulation to break if the user sets BreakOnAssertion to 2.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0 (note), 1 (warning), 2 (error), 3 (failure), 4 (fatal)
- **Default:** 3 (failure)

CheckPlusargs

This variable defines the simulator's behavior when encountering unrecognized plusargs.

- **Value Range:** 0 (ignores), 1 (issues warning, simulates while ignoring), 2 (issues error, exits)
- **Default:** 0 (ignores)

CheckpointCompressMode

This variable specifies that checkpoint files are written in compressed format

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** 0, 1
- **Default:** on (1)

CommandHistory

This variable specifies the name of a file in which to store the Main window command history.

- **Value Range:** any valid filename
- **Default:** commented out (;)

ConcurrentFileLimit

This variable controls the number of VHDL files open concurrently. This number should be less than the current limit setting for max file descriptors.

- **Value Range:** any positive integer or 0 (unlimited)
- **Default:** 40

CoverExcludeDefault

This variable excludes code coverage data collection from the default branch of case statements.

- **Value Range:** 0, 1
- **Default:** 0

CoverGenerate

This variable enables code coverage inside the top level of generate blocks.

- **Value Range:** 0, 1
- **Default:** 0

DatasetSeparator

This variable specifies the dataset separator for fully-rooted contexts, for example:

```
sim:/top
```

The argument to DatasetSeparator must not be the same character as PathSeparator

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** :

DefaultForceKind

This variable defines the kind of force used when not otherwise specified.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** freeze, drive, or deposit
- **Default:** drive, for resolved signals; freeze, for unresolved signals

DefaultRadix

This variable specifies a numeric radix may be specified as a name or number. For example, you can specify binary as “binary” or “2” or octal as “octal” or “8”.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii
- **Default:** symbolic

DefaultRestartOptions

This variable sets the default behavior for the restart command

- **Value Range:** one or more of: -force, -noassertions, -nobreakpoint, -nofcovers, -nolist, -nolog, -nowave
- **Default:** commented out (;)

DelayFileOpen

This variable instructs the tool to open VHDL87 files on first read or write, else open files when elaborated.

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** 0, 1
- **Default:** off (0)

GenerateFormat

This variable controls the format of a generate statement label. Do not enclose the argument in quotation marks.

- **Value Range:** Any non-quoted string containing at a minimum a %s followed by a %d

- **Default:** %s__%d

GlobalSharedObjectsList

This variable instruct the tool to load the specified PLI/FLI shared objects with global symbol visibility.

- **Value Range:** comma separated list of filenames
- **Default:** commented out (;)

IgnoreError

This variable instructs the tool to ignore VHDL assertion errors.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreFailure

This variable instructs the tool to ignore VHDL assertion failures.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreNote

This variable instructs the tool to ignore VHDL assertion notes.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IgnoreWarning

This variable instructs the tool to ignore VHDL assertion warnings.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0,1
- **Default:** off (0)

IterationLimit

This variable specifies a limit on simulation kernel iterations allowed without advancing time.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** positive integer
- **Default:** 5000

License

This variable controls the license file search.

- **Value Range:** one ore more of the following <license_option>, separated by spaces if using multiple entries. Refer also to the vsim <license_option>.

Table A-3.

license_option	**Description**
lnlonly	only use msimhdlsim and hdlsim
mixedonly	exclude single language licenses
nomgc	exclude MGC licenses
nolnl	exclude language neutral licenses
nomix	exclude msimhdlmix and hdlmix
nomti	exclude MTI licenses
noqueue	do not wait in license queue if no licenses are available
noslvhdl	exclude qhsimvh and vsim
noslvlog	exclude qhsimvl and vsimvlog
plus	only use PLUS license
vlog	only use VLOG license
vhdl	only use VHDL license

- **Default:** search all licenses

LockedMemory

For HP-UX 10.2 use only. This variable enables memory locking to speed up large designs (> 500mb memory footprint)

- **Value Range:** positive integer in units of MB.
- **Default:** disabled

NumericStdNoWarnings

This variable disables warnings generated within the accelerated numeric_std and numeric_bit packages.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0, 1
- **Default:** off (0)

PathSeparator

This variable specifies the character used for hierarchical boundaries of HDL modules. This variable does not affect file system paths. The argument to PathSeparator must not be the same character as DatasetSeparator.

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** /

Resolution

This variable specifies the simulator resolution. The argument must be less than or equal to the UserTimeUnit and must not contain a space between value and units, for example:

```
Resoultion = 10fs
```

You can override this value with the -t argument to vsim. You should set a smaller resolution if your delays get truncated.

- **Value Range:** fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100
- **Default:**

RunLength

This variable specifies the default simulation length in units specified by the UserTimeUnit variable

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** positive integer
- **Default:** 100

ScTimeUnit

This variable sets the default time unit for SystemC simulations.

- **Value Range:** fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100
- **Default:** 1 ns

SignalSpyPathSeparator

This variable specifies a unique path separator for the Signal Spy functions. The argument to SignalSpyPathSeparator must not be the same character as DatasetSeparator.

- **Value Range:** any character except those with special meaning, such as \, {, }, etc.
- **Default:** /

ShowUnassociatedScNameWarning

This variable instructs the tool to display unassociated SystemC name warnings.

- **Value Range:** 0, 1
- **Default:** off (1)

ShowUndebuggableScTypeWarning

This variable instructs the tool to display undebuggable SystemC type warnings.

- **Value Range:** 0, 1
- **Default:** on (1)

Startup

This variable specifies a simulation startup macro. Refer to the do command

- **Value Range:** = do <DO filename>; any valid macro (do) file

- **Default:** commented out (;)

StdArithNoWarnings

This variable suppresses warnings generated within the accelerated Synopsys std_arith packages.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** 0, 1
- **Default:** off (0)

ToggleMaxIntValues

This variable sets the maximum number of VHDL integer values to record with toggle coverage.

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** positive integer
- **Default:** 100

TranscriptFile

This variable specifies a file for saving command transcript. You can specify environment variables in the pathname.

- **Value Range:** any valid filename
- **Default:** transcript

UnbufferedOutput

This variable controls VHDL and Verilog files open for write.

- **Value Range:** 0 (buffered), 1 (unbuffered)
- **Default:** 0

UseCsupV2

Applies only to HP-UX 11.00 and when you compiled FLI/PLI/VPI C++ code with the -AA option for aCC.

This variable instructs **vsim** to use */usr/lib/libCsup_v2.sl* for shared object loading.

- **Value Range:** 0, 1
- **Default:** off (0)

UserTimeUnit

This variable specifies scaling for the Wave window and the default time units to use for commands such as force and run. You should generally set this variable to default, in which case it takes the value of the Resolution variable.

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** fs, ps, ns, us, ms, sec, or default
- **Default:** default

Veriuser

This variable specifies a list of dynamically loadable objects for Verilog PLI/VPI applications.

- **Value Range:** one or more valid shared object names
- **Default:** commented out (;)

VoptAutoSDFCompile

This variable controls whether or not SDF files are compiled as part of the design-wide optimizations vopt performs when automatic optimizations are performed. Valid arguments:

- **1** – Default. SDF files are compiled as part of design-wide optimizations.
- **0** – SDF files are not compiled in with the design.

VoptCoverageOptions

This variable turns off optimizations that interfere with collection of coverage statistics.

- **Default** – +acc=lprnb -opt=-merge

VoptFlow

This variable controls whether the tool operates in optimized mode or full visibility mode. Valid arguments:

- **1** -- Default. vopt is invoked automatically on design, the design is fully optimized.
- **0** -- Design is compiled and simulated without optimizations, maintaining full visibility.

Refer to Optimizing Designs with vopt for more details on optimization and vopt.

WarnConstantChange

This variable controls whether a warning is issued when the change command changes the value of a VHDL constant or generic.

- **Value Range:** 0, 1
- **Default:** on (1)

WaveSignalNameWidth

This variable controls the number of visible hierarchical regions of a signal name shown in the Wave Window.

- **Value Range:** 0 (display full name), positive integer (display corresponding level of hierarchy)
- **Default:** 0

WLFCacheSize

This variable sets the number of megabytes for the WLF reader cache; WLF reader caching caches blocks of the WLF file to reduce redundant file I/O

- **Value Range:** positive integer
- **Default:** 0

WLFCollapseMode

This variable controls when the WLF file records values.

- **Value Range:** 0 (every change of logged object), 1 (end of each delta step), 2 (end of simulator time step)
- **Default:** 1

WLFCompress

This variable enables WLF file compression.

- **Value Range:** 0, 1
- **Default:** 1 (on)

WLFDeleteOnQuit

This variable specifies whether a WLF file should be deleted when the simulation ends.

- **Value Range:** 0, 1

- **Default:** 0 (do not delete)

WLFFilename

This variable specifies the default WLF file name.

You can set this variable interactively with the Tcl set Command Syntax.

- **Value Range:** 0, 1
- **Default:** vsim.wlf

WLFOptimize

This variable specifies whether the viewing of waveforms is optimized.

- **Value Range:** 0, 1
- **Default:** 1 (on)

WLFSaveAllRegions

This variable specifies the regions to save in the WLF file.

- **Value Range:** 0 (only regions containing logged signals), 1 (all design hierarchy)
- **Default:** 0

WLFSizeLimit

This variable limits the WLF file by size (as closely as possible) to the specified number of megabytes; if both size and time limits are specified the most restrictive is used.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

- **Value Range:** any positive integer in units of MB or 0 (unlimited)
- **Default:** 0 (unlimited)

WLFTimeLimit

This variable limits the WLF file by time (as closely as possible) to the specified amount of time. If both time and size limits are specified the most restrictive is used.

You can set this variable interactively with the Tcl set Command Syntax or in the GUI; refer to Setting Simulator Control Variables With The GUI.

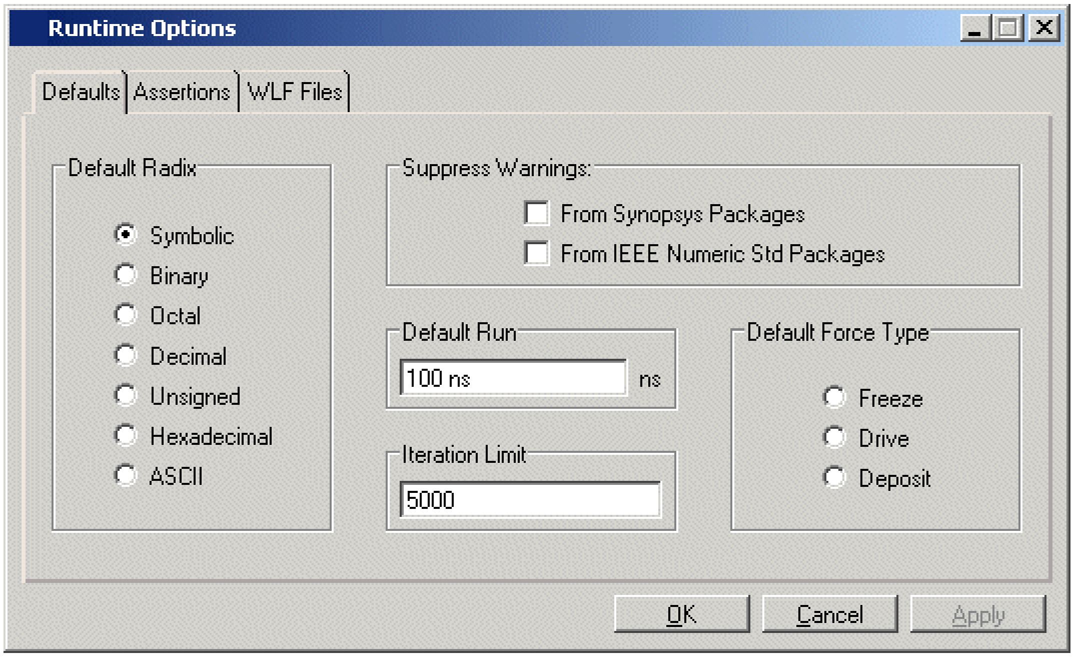
- **Value Range:** any positive integer or 0 (unlimited)

- **Default:** 0 (unlimited)

Setting Simulator Control Variables With The GUI

Changes made in the **Runtime Options** dialog are written to the active *modelsim.ini* file, if it is writable, and affect the current session as well as all future sessions. If the file is read-only, the changes affect only the current session. The **Runtime Options** dialog is accessible by selecting **Simulate > Runtime Options** in the Main window. The dialog contains three tabs - Defaults, Assertions, and WLF Files.

The Defaults tab includes these options:



Default Radix

● Symbolic
○ Binary
○ Octal
○ Decimal
○ Unsigned
○ Hexadecimal
○ ASCII

Suppress Warnings:

□ From Synopsys Packages
□ From IEEE Numeric Std Packages

Default Run

100 ns ns

Iteration Limit

5000

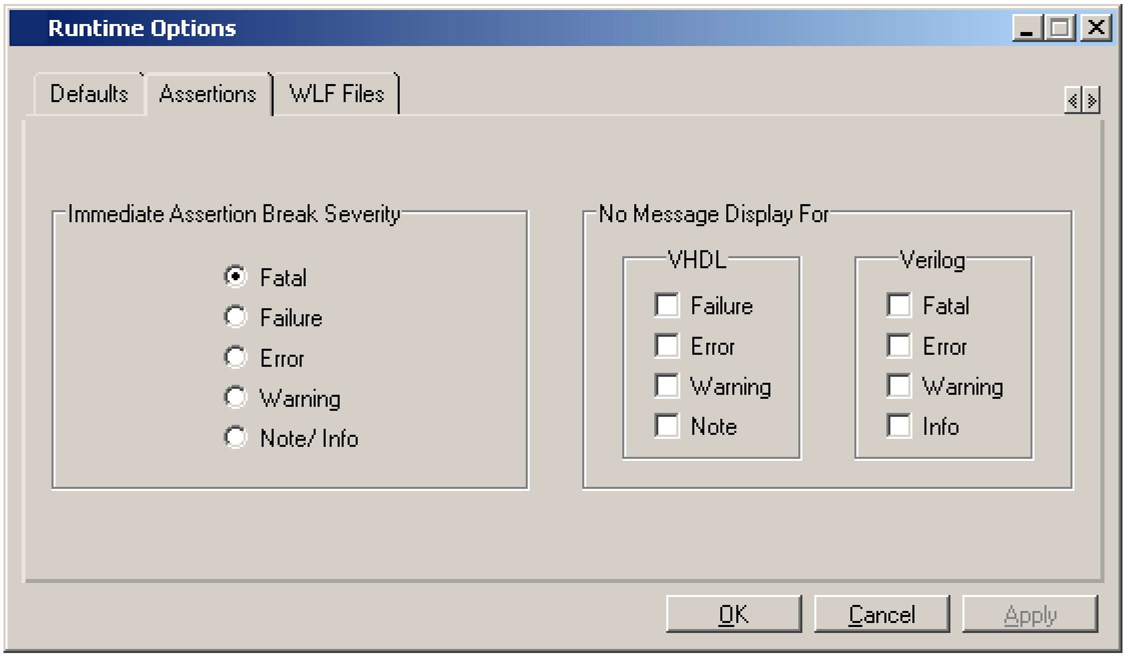
Default Force Type

○ Freeze
○ Drive
○ Deposit

- **Default Radix** — Sets the default radix for the current simulation run. You can also use the radix command to set the same temporary default. The chosen radix is used for all commands (force, examine, change are examples) and for displayed values in the Objects, Locals, Dataflow, List, and Wave windows. The corresponding *modelsim.ini* variable is DefaultRadix.
- **Suppress Warnings**
  - Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. The corresponding *modelsim.ini* variable is StdArithNoWarnings.
  - Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. The corresponding *modelsim.ini* variable is NumericStdNoWarnings.
- **Default Run —** Sets the default run length for the current simulation. The corresponding *modelsim.ini* variable is RunLength.

- **Iteration Limit —** Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. The corresponding *modelsim.ini* variable is IterationLimit.
- **Default Force Type** — Selects the default force type for the current simulation. The corresponding *modelsim.ini* variable is DefaultForceKind.

The Assertions tab includes these options:



Immediate Assertion Break Severity

● Fatal
○ Failure
○ Error
○ Warning
○ Note/ Info

No Message Display For

VHDL

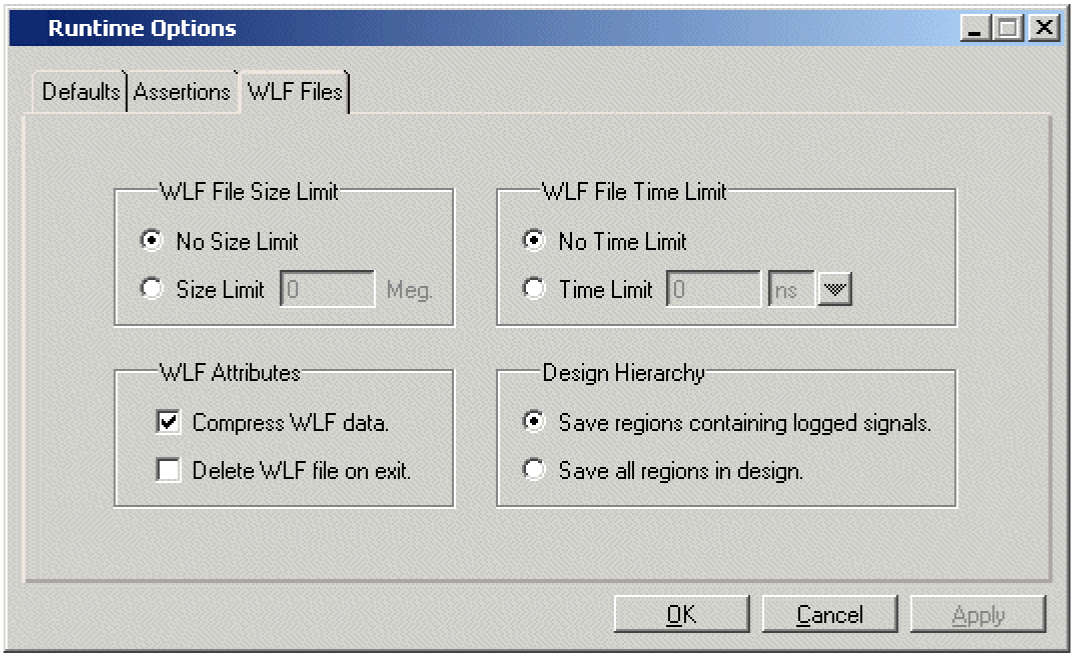
□ Failure
□ Error
□ Warning
□ Note

Verilog

□ Fatal
□ Error
□ Warning
□ Info

- **No Message Display For -VHDL** — Selects the VHDL assertion severity for which messages will not be displayed (even if break on assertion is set for that severity). Multiple selections are possible. The corresponding *modelsim.ini* variables are IgnoreFailure, IgnoreError, IgnoreWarning, and IgnoreNote.

The WLF Files tab includes these options:



WLF File Size Limit
- ● No Size Limit
- ○ Size Limit 0 Meg.

WLF File Time Limit
- ● No Time Limit
- ○ Time Limit 0 ns

WLF Attributes
- ☑ Compress WLF data.
- □ Delete WLF file on exit.

Design Hierarchy
- ● Save regions containing logged signals.
- ○ Save all regions in design.

- **WLF File Size Limit** — Limits the WLF file by size (as closely as possible) to the specified number of megabytes. If both size and time limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is WLFSizeLimit.
- **WLF File Time Limit** — Limits the WLF file by size (as closely as possible) to the specified amount of time. If both time and size limits are specified, the most restrictive is used. Setting it to 0 results in no limit. The corresponding *modelsim.ini* variable is WLFTimeLimit.
- **WLF Attributes** — Specifies whether to compress WLF files and whether to delete the WLF file when the simulation ends. You would typically only disable compression for troubleshooting purposes. The corresponding *modelsim.ini* variables are WLFCompress for compression and WLFDeleteOnQuit for WLF file deletion.
- **Design Hierarchy** — Specifies whether to save all design hierarchy in the WLF file or only regions containing logged signals. The corresponding *modelsim.ini* variable is WLFSaveAllRegions.

Logic Modeling Variables

Logic Modeling SmartModels and Hardware Modeler Interface

ModelSim's interface with Logic Modeling's SmartModels and hardware modeler are specified in the **[lmc]** section of the *INI*/MPF file; for more information see VHDL SmartModel Interface and VHDL Hardware Model Interface respectively.

Message System Variables

The message system variables (located under the [msg_system] heading) help you identify and troubleshoot problems while using the application. See also Message System.

error

This variable changes the severity of the listed message numbers to "error". Refer to Changing Message Severity Level for more information.

- **Value Range:** list of message numbers
- **Default:** none

fatal

This variable changes the severity of the listed message numbers to "fatal". Refer to Changing Message Severity Level for more information.

- **Value Range:** list of message numbers
- **Default:** none

note

This variable changes the severity of the listed message numbers to "note". Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

suppress

This variable suppresses the listed message numbers. Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

warning

This variable changes the severity of the listed message numbers to "warning". Refer to Changing Message Severity Level for more information

- **Value Range:** list of message numbers
- **Default:** none

msgmode

This variable controls where the simulator outputs elaboration and runtime messages. Refer to the section “Message Viewer” for more information.

- Value Range: tran (transcript only), wlf (wlf file only), both
- Default: both

Reading Variable Values From the INI File

You can read values from the *modelsim.ini* file with the following function:

```
GetPrivateProfileString <section> <key> <defaultValue>
```

Reads the string value for the specified variable in the specified section. Optionally provides a default value if no value is present.

Setting Tcl variables with values from the *modelsim.ini* file is one use of these Tcl functions. For example,

```
set MyCheckpointCompressMode [GetPrivateProfileString vsim
                                       CheckpointCompressMode 1]
set PrefMain(file) [GetPrivateProfileString vsim TranscriptFile ""]
```

Commonly Used INI Variables

Several of the more commonly used *modelsim.ini* variables are further explained below.

Common Environment Variables

You can use environment variables in your initialization files. Use a dollar sign ($) before the environment variable name. For example:

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is the name of the directory from which the VCOM or VLOG compilers or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

Hierarchical Library Mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the ModelSim tools don't find a mapping in the *modelsim.ini* file, then they will search only the library section of the initialization file specified by the "others" clause. For example:

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Since the file referred to by the "others" clause may itself contain an "others" clause, you can use this feature to chain a set of hierarchical INI files for library mappings.

Creating a Transcript File

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the ModelSim history.

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

You can disable the creation of the transcript file by using the following ModelSim command immediately after ModelSim starts:

```
transcript file ""
```

Using a Startup File

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs ModelSim to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the do command for additional information on creating do files.

Turning Off Assertion Messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Turning off Warnings from Arithmetic Packages

You can disable warnings from the Synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

These variables can also be set interactively using the Tcl set Command Syntax. This capability provides an answer to a common question about disabling warnings at time 0. You might enter commands like the following in a DO file or at the ModelSim prompt:

```
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Force Command Defaults

The **force** command has **-freeze**, **-drive**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

Restart Command Defaults

The **restart** command has **-force**, **-nobreakpoint**, **-nofcovers**, **-nolist**, **-nolog**, and **-nowave** options. You can set any of these as defaults by entering the following line in the *modelsim.ini* file:

```
DefaultRestartOptions = <options>
```

where <options> can be one or more of -force, -nobreakpoint, -nofcovers, -nolist, -nolog, and -nowave.

Example:

```
DefaultRestartOptions = -nolog -force
```

VHDL Standard

You can specify which version of the 1076 Std ModelSim follows by default using the VHDL93 variable:

```
[vcom]
; VHDL93 variable selects language version as the default.
; Default is VHDL-2002.
; Value of 0 or 1987 for VHDL-1987.
; Value of 1 or 1993 for VHDL-1993.
; Default or value of 2 or 2002 for VHDL-2002.
VHDL93 = 2002
```

Opening VHDL Files

You can delay the opening of VHDL files with an entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the **DelayFileOpen** option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

Variable Precedence

Note that some variables can be set in a .modelsim file (Registry in Windows) or a .ini file. A variable set in the .modelsim file takes precedence over the same variable set in a .ini file. For example, assume you have the following line in your *modelsim.ini* file:

```
TranscriptFile = transcript
```

And assume you have the following line in your *.modelsim* file:

```
set PrefMain(file) {}
```

In this case the setting in the *.modelsim* file overrides that in the *modelsim.ini* file, and a transcript file will not be produced.

Simulator State Variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within ModelSim DO files (macros). The variables are referenced in commands by prefixing the name with a dollar sign ($).

argc

This variable returns the total number of parameters passed to the current macro.

architecture

This variable returns the name of the top-level architecture currently being simulated; for an optimized Verilog module, returns architecture name; for a configuration or non-optimized Verilog module, this variable returns an empty string.

configuration

This variable returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration.

delta

This variable returns the number of the current simulator iteration.

entity

This variable returns the name of the top-level VHDL entity or Verilog module currently being simulated.

library

This variable returns the library name for the current region.

MacroNestingLevel

This variable returns the current depth of macro call nesting.

n

This variable represents a macro parameter, where n can be an integer in the range 1-9.

Now

This variable always returns the current simulation time with time units (e.g., 110,000 ns) Note: will return a comma between thousands.

now

This variable when time resolution is a unary unit (i.e., 1ns, 1ps, 1fs): returns the current simulation time without time units (e.g., 100000) when time resolution is a multiple of the unary unit (i.e., 10ns, 100ps, 10fs): returns the current simulation time with time units (e.g. 110000 ns) Note: will not return comma between thousands.

resolution

This variable returns the current simulation time resolution.

Referencing Simulator State Variables

Variable values may be referenced in simulator commands by preceding the variable name with a dollar sign ($). For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 ps 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \$now will not be interpreted as the current simulator time.

Special Considerations for the now Variable

For the when command, special processing is performed on comparisons involving the **now** variable. If you specify "when {$now=100}...", the simulator will stop at time 100 regardless of the multiplier applied to the time resolution.

You must use 64-bit time operators if the time value of **now** will exceed 2147483647 (the limit of 32-bit numbers). For example:

```
if { [gtTime $now 2us] } {
.
.
.
```

See Simulator Tcl Time Commands for details on 64-bit time operators.

Appendix B
Location Mapping

Pathnames to source files are recorded in libraries by storing the working directory from which the compile is invoked and the pathname to the file as specified in the invocation of the compiler. The pathname may be either a complete pathname or a relative pathname.

Referencing Source Files with Location Maps

ModelSim tools that reference source files from the library locate a source file as follows:

- If the pathname stored in the library is complete, then this is the path used to reference the file.
- If the pathname is relative, then the tool looks for the file relative to the current working directory. If this file does not exist, then the path relative to the working directory stored in the library is used.

This method of referencing source files generally works fine if the libraries are created and used on a single system. However, when multiple systems access a library across a network, the physical pathnames are not always the same and the source file reference rules do not always work.

Using Location Mapping

Location maps are used to replace prefixes of physical pathnames in the library with environment variables. The location map defines a mapping between physical pathname prefixes and environment variables.

ModelSim tools open the location map file on invocation if the MGC_LOCATION_MAP environment variable is set. If MGC_LOCATION_MAP is not set, ModelSim will look for a file named *"mgc_location_map"* in the following locations, in order:

- the current directory
- your home directory
- the directory containing the ModelSim binaries
- the ModelSim installation directory

Use these two steps to map your files:

1. Set the environment variable MGC_LOCATION_MAP to the path to your location map file.
2. Specify the mappings from physical pathnames to logical pathnames:

```
$SRC
/home/vhdl/src
/usr/vhdl/src

$IEEE
/usr/modeltech/ieee
```

Pathname Syntax

The logical pathnames must begin with *$* and the physical pathnames must begin with */*. The logical pathname is followed by one or more equivalent physical pathnames. Physical pathnames are equivalent if they refer to the same physical directory (they just have different pathnames on different systems).

How Location Mapping Works

When a pathname is stored, an attempt is made to map the physical pathname to a path relative to a logical pathname. This is done by searching the location map file for the first physical pathname that is a prefix to the pathname in question. The logical pathname is then substituted for the prefix. For example, "/usr/vhdl/src/test.vhd" is mapped to "$SRC/test.vhd". If a mapping can be made to a logical pathname, then this is the pathname that is saved. The path to a source file entry for a design unit in a library is a good example of a typical mapping.

For mapping from a logical pathname back to the physical pathname, ModelSim expects an environment variable to be set for each logical pathname (with the same name). ModelSim reads the location map file when a tool is invoked. If the environment variables corresponding to logical pathnames have not been set in your shell, ModelSim sets the variables to the first physical pathname following the logical pathname in the location map. For example, if you don't set the SRC environment variable, ModelSim will automatically set it to "/home/vhdl/src".

Mapping with TCL Variables

Two Tcl variables may also be used to specify alternative source-file paths; SourceDir and SourceMap. You would define these variables in a *modelsim.tcl* file. See the The modelsim.tcl File for details.

Appendix C
Error and Warning Messages

Message System

The ModelSim message system helps you identify and troubleshoot problems while using the application. The messages display in a standard format in the Transcript pane. Accordingly, you can also access them from a saved transcript file (see Saving the Transcript File for more details).

Message Format

The format for the messages is:

```
** <SEVERITY LEVEL>: ([<Tool>[-<Group>]]-<MsgNum>) <Message>
```

SEVERITY LEVEL may be one of the following:

Table C-1.

severity level	**meaning**
Note	This is an informational message.
Warning	There may be a problem that will affect the accuracy of your results.
Error	The tool cannot complete the operation.
Fatal	The tool cannot complete execution.
INTERNAL ERROR	This is an unexpected error that should be reported to your support representative.

Tool indicates which ModelSim tool was being executed when the message was generated. For example tool could be **vcom**, **vdel**, **vsim**, etc.

Group indicates the topic to which the problem is related. For example group could be FLI, PLI, VCD, etc.

Example

```
# ** Error: (vsim-PLI-3071) ./src/19/testfile(77): $fdumplimit : Too few
arguments.
```

Getting More Information

Each message is identified by a unique MsgNum id. You can access additional information about a message using the unique id and the verror command. For example:

```
% verror 3071
Message # 3071:
Not enough arguments are being passed to the specified system task or
function.
```

Changing Message Severity Level

You can change the severity of or suppress notes, warnings, and errors that come from **vcom**, **vlog**, and **vsim**. You cannot change the severity of or suppress Fatal or Internal messages.

There are two ways to modify the severity of or suppress notes, warnings, and errors:

- Use the -error, -fatal, -note, -suppress, and -warning arguments to sccom, vcom, vlog, vopt, or vsim. See the command descriptions in the Reference Manual for details on those arguments.
- Set a permanent default in the [msg_system] section of the *modelsim.ini* file. See Control Variables Located in INI Files for more information.

Suppressing Warning Messages

You can suppress some warning messages. For example, you may receive warning messages about unbound components about which you are not concerned.

Suppressing VCOM Warning Messages

Use the -nowarn <number> argument to vcom to suppress a specific warning message. For example:

```
vcom -nowarn 1
```

Suppresses unbound component warning messages.

Alternatively, warnings may be disabled for all compiles via the *modelsim.ini* file (see Verilog Compiler Control Variables).

The warning message numbers are:

```
1 = unbound component
2 = process without a wait statement
3 = null range
4 = no space in time literal
5 = multiple drivers on unresolved signal
6 = compliance checks
7 = optimization messages
8 = lint checks
9 = signal value dependency at elaboration
10 = VHDL93 constructs in VHDL87 code
14 = locally static error deferred until simulation run
```

These numbers are category-of-warning message numbers. They are unrelated to vcom arguments that are specified by numbers, such as **vcom -87** – which disables support for VHDL-1993 and 2002.

Suppressing VLOG Warning Messages

Use the +nowarn<CODE> argument to vlog to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vlog +nowarnDECAY
```

Suppresses decay warning messages.

Suppressing VSIM Warning Messages

Use the +nowarn<CODE> argument to vsim to suppress a specific warning message. Warnings that can be disabled include the <CODE> name in square brackets in the warning message. For example:

```
vsim +nowarnTFMPC
```

Suppresses warning messages about too few port connections.

Exit Codes

The table below describes exit codes used by ModelSim tools.

Table C-2.

Exit code	Description
0	Normal (non-error) return
1	Incorrect invocation of tool
2	Previous errors prevent continuing
3	Cannot create a system process (execv, fork, spawn, etc.)

Table C-2.

Exit code	Description
4	Licensing problem
5	Cannot create/open/find/read/write a design library
6	Cannot create/open/find/read/write a design unit
7	Cannot open/read/write/dup a file (open, lseek, write, mmap, munmap, fopen, fdopen, fread, dup2, etc.)
8	File is corrupted or incorrect type, version, or format of file
9	Memory allocation error
10	General language semantics error
11	General language syntax error
12	Problem during load or elaboration
13	Problem during restore
14	Problem during refresh
15	Communication problem (Cannot create/read/write/close pipe/socket)
16	Version incompatibility
19	License manager not found/unreadable/unexecutable (vlm/mgvlm)
22	SystemC link error
23	SystemC DPI internal error
42	Lost license
43	License read/write failure
44	Modeltech daemon license checkout failure #44
45	Modeltech daemon license checkout failure #45
90	Assertion failure (SEVERITY_QUIT)
99	Unexpected error in tool
100	GUI Tcl initialization failure
101	GUI Tk initialization failure
102	GUI IncrTk initialization failure
111	X11 display error
202	Interrupt (SIGINT)
204	Illegal instruction (SIGILL)

Table C-2.

Exit code	Description
205	Trace trap (SIGTRAP)
206	Abort (SIGABRT)
208	Floating point exception (SIGFPE)
210	Bus error (SIGBUS)
211	Segmentation violation (SIGSEGV)
213	Write on a pipe with no reader (SIGPIPE)
214	Alarm clock (SIGALRM)
215	Software termination signal from kill (SIGTERM)
216	User-defined signal 1 (SIGUSR1)
217	User-defined signal 2 (SIGUSR2)
218	Child status change (SIGCHLD)
230	Exceeded CPU limit (SIGXCPU)
231	Exceeded file size limit (SIGXFSZ)

Miscellaneous Messages

This section describes miscellaneous messages which may be associated with ModelSim.

- Compilation of DPI Export TFs Error
  - Message Text

    ```
    # ** Fatal: (vsim-3740) Can't locate a C compiler for compilation of
                                                    DPI export tasks/functions.
    ```
  - Meaning

    ModelSim was unable to locate a C compiler to compile the DPI exported tasks or functions in your design.
  - Suggested Action

    Make sure that a C compiler is visible from where you are running the simulation.
- Empty port name warning
  - Message text

    ```
    # ** WARNING: [8] <path/file_name>:
    empty port name in port list.
    ```
  - Meaning

ModelSim reports these warnings if you use the **-lint** argument to vlog. It reports the warning for any NULL module ports.

- Suggested action

  If you wish to ignore this warning, do not use the **-lint** argument.

- Lock message
  - Message text

    ```
    waiting for lock by user@user. Lockfile is <library_path>/_lock
    ```
  - Meaning

    The *_lock* file is created in a library when you begin a compilation into that library, and it is removed when the compilation completes. This prevents simultaneous updates to the library. If a previous compile did not terminate properly, ModelSim may fail to remove the *_lock* file.
  - Suggested action

    Manually remove the *_lock* file after making sure that no one else is actually using that library.
- Metavalue detected warning
  - Message text

    ```
    Warning: NUMERIC_STD.">": metavalue detected, returning FALSE
    ```
  - Meaning

    This warning is an assertion being issued by the IEEE **numeric_std** package. It indicates that there is an 'X' in the comparison.
  - Suggested action

    The message does not indicate which comparison is reporting the problem since the assertion is coming from a standard package. To track the problem, note the time the warning occurs, restart the simulation, and run to one time unit before the noted time. At this point, start stepping the simulator until the warning appears. The location of the blue arrow in a Source window will be pointing at the line following the line with the comparison.

    These messages can be turned off by setting the **NumericStdNoWarnings** variable to 1 from the command line or in the *modelsim.ini* file.
- Sensitivity list warning
  - Message text

    ```
    signal is read by the process but is not in the sensitivity list
    ```
  - Meaning

ModelSim outputs this message when you use the **-check_synthesis** argument to vcom. It reports the warning for any signal that is read by the process but is not in the sensitivity list.

- Suggested action

  There are cases where you may purposely omit signals from the sensitivity list even though they are read by the process. For example, in a strictly sequential process, you may prefer to include only the clock and reset in the sensitivity list because it would be a design error if any other signal triggered the process. In such cases, your only option is to not use the **-check_synthesis** argument.

- Tcl Initialization error 2

  - Message text

    ```
    Tcl_Init Error 2 : Can't find a usable Init.tcl in the following
          directories :
       ./../tcl/tcl8.3 .
    ```

  - Meaning

    This message typically occurs when the base file was not included in a Unix installation. When you install ModelSim, you need to download and install 3 files from the ftp site. These files are:

    ```
    modeltech-base.tar.gz
    modeltech-docs.tar.gz
    modeltech-<platform>.exe.gz
    ```

    If you install only the <platform> file, you will not get the Tcl files that are located in the base file.

    This message could also occur if the file or directory was deleted or corrupted.

  - Suggested action

    Reinstall ModelSim with all three files.

- Too few port connections

  - Message text

    ```
    # ** Warning (vsim-3017): foo.v(1422): [TFMPC] - Too few port
                                     connections. Expected 2, found 1.
    # Region: /foo/tb
    ```

  - Meaning

    This warning occurs when an instantiation has fewer port connections than the corresponding module definition. The warning doesn’t necessarily mean anything is wrong; it is legal in Verilog to have an instantiation that doesn’t connect all of the pins. However, someone that expects all pins to be connected would like to see such a warning.

Here are some examples of legal instantiations that will and will not cause the warning message.

Module definition:

```
module foo (a, b, c, d);
```

Instantiation that does not connect all pins but will not produce the warning:

```
foo inst1(e, f, g, ); // positional association
foo inst1(.a(e), .b(f), .c(g), .d()); // named association
```

Instantiation that does not connect all pins but will produce the warning:

```
foo inst1(e, f, g); // positional association
foo inst1(.a(e), .b(f), .c(g)); // named association
```

Any instantiation above will leave pin *d* unconnected but the first example has a placeholder for the connection. Here's another example:

```
foo inst1(e, , g, h);
foo inst1(.a(e), .b(), .c(g), .d(h));
```

- Suggested actions
    - Check that there is not an extra comma at the end of the port list. (e.g., model(a,b,)). The extra comma is legal Verilog and implies that there is a third port connection that is unnamed.
    - If you are purposefully leaving pins unconnected, you can disable these messages using the **+nowarnTFMPC** argument to vsim.

- VSIM license lost
    - Message text

      ```
      Console output:
      Signal 0 caught... Closing vsim vlm child.
      vsim is exiting with code 4
      FATAL ERROR in license manager

      transcript/vsim output:
      # ** Error: VSIM license lost; attempting to re-establish.
      #    Time: 5027 ns  Iteration: 2
      # ** Fatal: Unable to kill and restart license process.
      #    Time: 5027 ns  Iteration: 2
      ```
    - Meaning

      ModelSim queries the license server for a license at regular intervals. Usually these "License Lost" error messages indicate that network traffic is high, and communication with the license server times out.
    - Suggested action

Anything you can do to improve network communication with the license server will probably solve or decrease the frequency of this problem.

- Failed to find libswift entry

  o Message text

  ```
  ** Error: Failed to find LMC Smartmodel libswift entry in project
  file.
  # Fatal: Foreign module requested halt
  ```

  o Meaning

  ModelSim could not locate the **libswift** entry and therefore could not link to the Logic Modeling library.

  o Suggested action

  Uncomment the appropriate **libswift** entry in the [lmc] section of the *modelsim.ini* or project *.mpf* file. See VHDL SmartModel Interface for more information.

sccom Error Messages

This section describes sccom error messages which may be associated with ModelSim.

- Failed to load sc lib error: undefined symbol

  o Message text

  ```
  # ** Error: (vsim-3197) Load of
        "/home/cmg/newport2_systemc/chip/vhdl/work/systemc.so"
        failed:ld.so.1:
        /home/icds_nut/modelsim/5.8a/sunos5/vsimk:
        fatal: relocation error: file
        /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so:
        symbol_Z28host_respond_to_vhdl_requestPm:
        referenced symbol not found.
  # ** Error: (vsim-3676) Could not load shared library
        /home/cmg/newport2_systemc/chip/vhdl/work/systemc.so
        for SystemC module 'host_xtor'.
  ```

  o Meaning

  The causes for such an error could be:

  - missing symbol definition
  - bad link order specified in sccom -link
  - multiply defined symbols (see Multiple Symbol Definitions)

  o Suggested action

  - If the undefined symbol is a C function in your code or a library you are linking with, be sure that you declared it as an extern "C" function:

extern "C" void myFunc();

- The order in which you place the **-link** option within the **sccom -link** command is critical. Make sure you have used it appropriately. See sccom for syntax and usage information. See Misplaced -link Option for further explanation of error and correction.

- Multiply defined symbols

  o Message text

```
work/sc/gensrc/test_ringbuf.o: In function
          `test_ringbuf::clock_generator(void)':
work/sc/gensrc/test_ringbuf.o(.text+0x4): multiple definition of
          `test_ringbuf::clock_generator(void)'
work/sc/test_ringbuf.o(.text+0x4): first defined here
```

  o Meaning

  The most common type of error found during **sccom -link** operation is the multiple symbol definition error. This typically arises when the same global symbol is present in more than one *.o* file. Several causes are likely:

  - A common cause of multiple symbol definitions involves incorrect definition of symbols in header files. If you have an out-of-line function (one that isn't preceded by the "inline" keyword) or a variable defined (i.e., not just referenced or prototyped, but truly defined) in a *.h* file, you can't include that *.h* file in more than one *.cpp* file.

  - Another cause of errors is due to ModelSim's name association feature. The name association feature automatically generates *.cpp* files in the work library. These files "include" your header files. Thus, while it might appear as though you have included your header file in only one *.cpp* file, from the linker's point of view, it is included in multiple *.cpp* files.

  o Suggested action

  Make sure you don't have any out-of-line functions. Use the "inline" keyword. See Multiple Symbol Definitions.

Enforcing Strict 1076 Compliance with pedanticerrors

The optional **-pedanticerrors** argument to vcom enforces strict compliance to the IEEE 1076 LRM in the cases listed below. The default behavior for these cases is to issue an insuppressible warning message. If you compile with **-pedanticerrors**, the warnings change to an error, unless otherwise noted. Descriptions in quotes are actual warning/error messages emitted by **vcom**. As noted, in some cases you can suppress the warning using **-nowarn [level]**.

- Type conversion between array types, where the element subtypes of the arrays do not have identical constraints.

- "Extended identifier terminates at newline character (0xa)."
- "Extended identifier contains non-graphic character 0x%x."
- "Extended identifier \"%s\" contains no graphic characters."
- "Extended identifier \"%s\" did not terminate with backslash character."
- "An abstract literal and an identifier must have a separator between them."

  This is for forming physical literals, which comprise an optional numeric literal, followed by a separator, followed by an identifier (the unit name). Warning is level 4, which means "-nowarn 4" will suppress it.
- In VHDL 1993 or 2002, a subprogram parameter was declared using VHDL 1987 syntax (which means that it was a class VARIABLE parameter of a file type, which is the only way to do it in VHDL 1987 and is illegal in later VHDLs). Warning is level 10.
- "Shared variables must be of a protected type." Applies to VHDL 2002 only.
- Expressions evaluated during elaboration cannot depend on signal values. Warning is level 9.
- "Non-standard use of output port '%s' in PSL expression." Warning is level 11.
- "Non-standard use of linkage port '%s' in PSL expression." Warning is level 11.
- Type mark of type conversion expression must be a named type or subtype, it can't have a constraint on it.
- When the actual in a PORT MAP association is an expression, it must be a (globally) static expression. The port must also be of mode IN.
- The expression in the CASE and selected signal assignment statements must follow the rules given in 8.8 of the LRM. In certain cases we can relax these rules, but **-pedanticerrors** forces strict compliance.
- A CASE choice expression must be a locally static expression. We allow it to be only globally static, but **-pedanticerrors** will check that it is locally static. Same rule for selected signal assignment statement choices. Warning level is 8.
- When making a default binding for a component instantiation, ModelSim's non-standard search rules found a matching entity. VHDL 2002 LRM Section 5.2.2 spells out the standard search rules. Warning level is 1.
- Both FOR GENERATE and IF GENERATE expressions must be globally static. We allow non-static expressions unless **-pedanticerrors** is present.
- When the actual part of an association element is in the form of a conversion function call [or a type conversion], and the formal is of an unconstrained array type, the return type of the conversion function [type mark of the type conversion] must be of a

constrained array subtype. We relax this (with a warning) unless **-pedanticerrors** is present when it becomes an error.

- OTHERS choice in a record aggregate must refer to at least one record element.
- In an array aggregate of an array type whose element subtype is itself an array, all expressions in the array aggregate must have the same index constraint, which is the element's index constraint. No warning is issued; the presence of **-pedanticerrors** will produce an error.
- Non-static choice in an array aggregate must be the only choice in the only element association of the aggregate.
- The range constraint of a scalar subtype indication must have bounds both of the same type as the type mark of the subtype indication.
- The index constraint of an array subtype indication must have index ranges each of whose both bounds must be of the same type as the corresponding index subtype.
- When compiling VHDL 1987, various VHDL 1993 and 2002 syntax is allowed. Use **-pedanticerrors** to force strict compliance. Warnings are all level 10.

Appendix D
Verilog PLI/VPI/DPI

This appendix describes the ModelSim implementation of the Verilog PLI (Programming Language Interface), VPI (Verilog Procedural Interface) and SystemVerilog DPI (Direct Programming Interface). These three interfaces provide a mechanism for defining tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI (see Third Party PLI Applications). In addition, you may write your own PLI/VPI/DPI applications.

Implementation Information

ModelSim Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the **acc_handle_datapath()** routine. We did not implement the **acc_handle_datapath()** routine because the information it returns is more appropriate for a static timing analysis tool.

The VPI is partially implemented as defined in the IEEE Std 1364-2005. The list of currently supported functionality can be found in the following file:

<install_dir>/modeltech/docs/technotes/Verilog_VPI.note

ModelSim SystemVerilog implements DPI as defined in IEEE Std P1800-2005.

The IEEE Std 1364 is the reference that defines the usage of the PLI/VPI routines, and the IEEE Std P1800-2005 Language Reference Manual (LRM) defines the usage of DPI routines. This manual describes only the details of using the PLI/VPI/DPI with ModelSim Verilog and SystemVerilog.

g++ Compiler Support for use with PLI/VPI/DPI

We strongly encourage that unless you have a reason to do otherwise, you should use the built-in g++ compiler that is shipped with the ModelSim compiler to compile your C++ code. This is the version that has been tested and is supported for any given release.

Specifying Your Own g++ Compiler

If you must use a different g++ compiler, other than that shipped with ModelSim, you need to set a variable in your modelsim.ini file, as follows:

```
CppPath = /usr/bin/g++
```

to point to the desired g++ version.

Registering PLI Applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, ModelSim Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of s_tfcell structures. This structure is declared in the veriuser.h include file as follows:

```
typedef int (*p_tffn)();
typedef struct t_tfcell {
   short type;/* USERTASK, USERFUNCTION, or USERREALFUNCTION */
   short data;/* passed as data argument of callback function */
   p_tffn checktf;  /* argument checking callback function */
   p_tffn sizetf;   /* function return size callback function */
   p_tffn calltf;   /* task or function call callback function */
   p_tffn misctf;   /* miscellaneous reason callback function */
   char *tfname;/* name of system task or function */
      /* The following fields are ignored by ModelSim Verilog */
   int forwref;
   char *tfveritool;
   char *tferrmessage;
   int hash;
   struct t_tfcell *left_p;
   struct t_tfcell *right_p;
   char *namecell_p;
   int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications don't use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with $). The remaining fields are not used by ModelSim Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF() for each system task or function defined. The mti_RegisterUserTF() function is declared in veriuser.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation because the simulator de-references the usertf pointer to call the callback functions. We recommend that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
   {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
   {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
   {0}  /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
   p_tfcell usertf = veriusertfs;
   while (usertf->type)
      mti_RegisterUserTF(usertf++);
}
```

It is an error if a PLI shared library does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see Compiling and Linking C Applications for PLI/VPI/DPI). The PLI applications are specified as follows (note that on a Windows platform the file extension would be .dll):

- As a list in the Veriuser entry in the *modelsim.ini* file:

  Veriuser = pliapp1.so pliapp2.so pliappn.so

- As a list in the PLIOBJS environment variable:

  % setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"

- As a -pli argument to the simulator (multiple arguments are allowed):

  -pli pliapp1.so -pli pliapp2.so -pli pliappn.so

The various methods of specifying PLI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

Registering VPI Applications

Each VPI application must register its system tasks and functions and its callbacks with the simulator. To accomplish this, one or more user-created registration routines must be called at simulation startup. Each registration routine should make one or more calls to vpi_register_systf() to register user-defined system tasks and functions and vpi_register_cb() to register callbacks. The registration routines must be placed in a table named vlog_startup_routines so that the simulator can find them. The table must be terminated with a 0 entry.

Example D-1. VPI Application Registration

```
PLI_INT32 MyFuncCalltf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncCompiletf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyFuncSizetf( PLI_BYTE8 *user_data )
{ ... }
PLI_INT32 MyEndOfCompCB( p_cb_data cb_data_p )
{ ... }
PLI_INT32 MyStartOfSimCB( p_cb_data cb_data_p )
{ ... }
void RegisterMySystfs( void )
  {

      vpiHandle tmpH;
      s_cb_data callback;
      s_vpi_systf_data systf_data;

      systf_data.type        = vpiSysFunc;
      systf_data.sysfunctype = vpiSizedFunc;
      systf_data.tfname      = "$myfunc";
      systf_data.calltf      = MyFuncCalltf;
      systf_data.compiletf   = MyFuncCompiletf;
      systf_data.sizetf      = MyFuncSizetf;
      systf_data.user_data   = 0;
      tmpH = vpi_register_systf( &systf_data );
      vpi_free_object(tmpH);

      callback.reason    = cbEndOfCompile;
      callback.cb_rtn    = MyEndOfCompCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);

      callback.reason    = cbStartOfSimulation;
      callback.cb_rtn    = MyStartOfSimCB;
      callback.user_data = 0;
      tmpH = vpi_register_cb( &callback );
      vpi_free_object(tmpH);
  }

void (*vlog_startup_routines[ ] ) () = {
   RegisterMySystfs,
      0    /* last entry must be 0 */
};
```

Loading VPI applications into the simulator is the same as described in Registering PLI Applications.

Using PLI and VPI Together

PLI and VPI applications can co-exist in the same application object file. In such cases, the applications are loaded at startup as follows:

- If an init_usertfs() function exists, then it is executed and only those system tasks and functions registered by calls to mti_RegisterUserTF() will be defined.
- If an init_usertfs() function does not exist but a veriusertfs table does exist, then only those system tasks and functions listed in the veriusertfs table will be defined.
- If an init_usertfs() function does not exist and a veriusertfs table does not exist, but a vlog_startup_routines table does exist, then only those system tasks and functions and callbacks registered by functions in the vlog_startup_routines table will be defined.

As a result, when PLI and VPI applications exist in the same application object file, they must be registered in the same manner. VPI registration functions that would normally be listed in a vlog_startup_routines table can be called from an init_usertfs() function instead.

Registering DPI Applications

DPI applications do not need to be registered. However, each DPI imported or exported task or function must be identified using SystemVerilog ‘import “DPI-C”’ or ‘export “DPI-C”’syntax. Examples of the syntax follow:

```
export "DPI-C" task t1;
task t1(input int i, output int o);
.
.
.
end task
import "DPI-C" function void f1(input int i, output int o);
```

Your code must provide imported functions or tasks, compiled with an external compiler. An imported task must return an int value, "1" indicating that it is returning due to a disable, or "0" indicating otherwise.

These imported functions or objects may then be loaded as a shared library into the simulator with either the command line option **-sv_lib <lib>** or **-sv_liblist <bootstrap_file>**. For example,

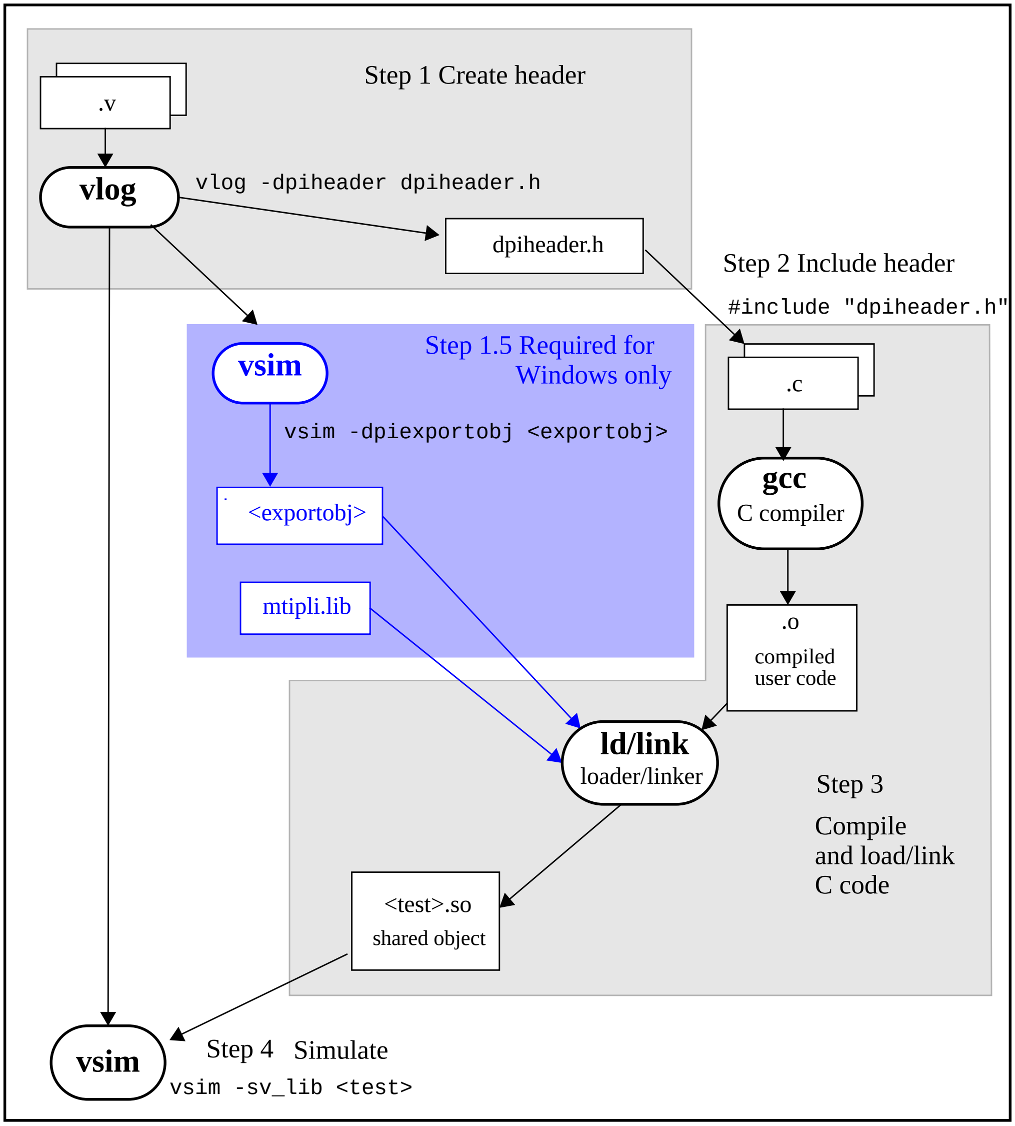
```
vlog dut.v
gcc -shared -o imports.so imports.c
vsim -sv_lib imports top -do <do_file>
```

The **-sv_lib** option specifies the shared library name, without an extension. A file extension is added by the tool, as appropriate to your platform. For a list of file extensions accepted by platform, see DPI File Loading.

You can also use the command line options **-sv_root** and **-sv_liblist** to control the process for loading imported functions and tasks. These options are defined in the IEEE Std P1800-2005 LRM.

DPI Use Flow

Correct use of ModelSim DPI depends on the flow presented in this section.



1. Run vlog to generate a *dpiheader.h* file.

   This file defines the interface between C and ModelSim for exported and imported tasks and functions. Though the *dpiheader.h* is a user convenience file rather than requirement, including *dpiheader.h* in your C code can immediately solve problems caused by an improperly defined interface. An example command for creating the header file would be:

```
vlog -dpiheader <dpiheader>.h
```

Required for Windows only; Run a preliminary invocation of vsim with the **-dpiexportobj** argument.

Because of limitations with the linker/loader provided on Windows, this additional step is required. You must create the exported task/function compiled object file (*exportobj*) by running a preliminary vsim command, such as:

```
vsim -dpiexportobj exportobj
```

2. Include the *dpiheader.h* file in your C code.

   ModelSim recommends that any user DPI C code that accesses exported tasks/functions, or defines imported tasks/functions, will include the *dpiheader.h* file. This allows the C compiler to verify the interface between C and ModelSim.

3. Compile the C code into a shared object.

   Compile your code, providing any *.a* or other *.o* files required.

   For Windows users — In this step, the object file is bound into the *.dll* that you created using the **-dpiexportobj** argument.

4. Simulate the design.

   When simulating, specify the name of the imported DPI C shared object (according to the SystemVerilog LRM). For example:

   ```
   vsim -sv_lib <test>
   ```

When Your DPI Export Function is Not Getting Called

This issue can arise in your C code due to the way the C linker resolves symbols. It happens if a name you choose for a SystemVerilog export function happens to match a function name in a standard C library. In this case, your C compiler will bind calls to the function in that C library, rather than to the export function in the SystemVerilog simulator.

The symptoms of such a misbinding can be difficult to detect. Generally, the misbound function silently returns some unexpected or incorrect value.

To determine if you have this type of name aliasing problem, consult the C library documentation (either the online help or man pages).

DPI and the qverilog Command

The qverilog performs a single-step compilation, optimization, and simulation (see Platform Specific Information). You can specify C/C++ files on the **qverilog** command line, and the command will invoke the correct C/C++ compiler based on the file type passed. For example, you can enter the following command:

qverilog verilog1.v verilog2.v mydpicode.c

This command:

1. Compiles the Verilog files with vlog
2. Compiles and links the C/C++ file
3. Creates the shared object *qv_dpi.so* in the *work/_dpi* directory.
4. Invokes the vsim with the **-sv_lib** argument and the shared object created in step 3.

For **-ccflags** and **-ldflags**, **qverilog** does not check the validity of the option(s) you specify. The options are directly passed on to the compiler and linker, and if they are not valid, an error message is generated by the compiler/linker.

You can also specify C/C++ files and options with the **-f** argument, and they will be processed the same way as Verilog files and options in a **-f** file.

Note

If your design contains export tasks and functions, it is recommended that you use the classic simulation flow (vlog/vsim).

Platform Specific Information

For designs containing only DPI import tasks and functions (no exports), the simplified qverilog flow with C/C++ files on the command line works great on all platforms. On Win32/AIX, use the two step vlog/vsim flow for designs with export tasks and functions.

Simplified Import of FLI / PLI / C Library Functions

In addition to the traditional method of importing FLI / PLI / C library functions, a simplified method can be used: you can declare VPI and FLI functions as DPI-C imports. When you declare VPI and FLI functions as DPI-C imports, the DPI shared object is loaded at runtime automatically. Neither the C implementation of the import tf, nor the **-sv_lib** argument is required.

Also, on most platforms (see Platform Specific Information), you can declare most standard C library functions as DPI-C imports.

The following example is processed directly, without DPI C code:

```
package cmath;
    import "DPI-C" function real sin(input real x);
    import "DPI-C" function real sqrt(input real x);
endpackage

package fli;
    import "DPI-C" function mti_Cmd(input string cmd);
```

```
endpackage

module top;
    import cmath::*;
    import fli::*;
    int status, A;
    initial begin
        $display("sin(0.98) = %f", sin(0.98));
        $display("sqrt(0.98) = %f", sqrt(0.98));
        status = mti_Cmd("change A 123");
        $display("A = %1d, status = %1d", A, status);
    end
endmodule
```

To simulate, you would simply enter a command such as: **vsim <test>**.

Platform Specific Information

This feature is not supported on AIX.

On Windows, only FLI and PLI commands may be imported in this fashion. C library functions are not automatically importable. They must be wrapped in user DPI C functions, which are brought into the simulator using the **-sv_lib** argument.

Use Model for Read-Only Work Libraries

You may want to create the work library as a read-only entity, which enables multiple users to simultaneously share the design library at runtime. The steps are as follows:

- Windows and RS6000/RS64

  On these platforms, simply change the permissions on the design library to read only by issuing a command such as "chmod -R a-w <libname>". Do this after you have finished compiling with vlog/vcom and vopt.

- All Other Platforms

  If a design contains no DPI export tasks or functions, the work library can be changed by simply changing the permissions, as shown for win32 and rs6000/rs64 above.

  For designs that contain DPI export tasks and functions, and are not run on Windows or RS6000/RS64, by default vsim creates a shared object in directory *<libname>/_dpi*. This shared object is called *exportwrapper.so* (Linux and Solaris) or *exportwrapper.sl (*hp700, hppa64, and hpux_ia64). If you are using a read-only library, **vsim** must not create any objects in the library.

To prevent **vsim** from creating objects in the library at runtime, the **vsim -dpiexportobj** flow is available on all platforms. Use this flow after compilation, but before you start simulation using the design library.

An example command sequence on Linux would be:

```
vlib work
vlog -dpiheader dpiheader.h test.sv
gcc -shared -o test.so test.c
vsim -c -dpiexportobj work/_dpi/exportwrapper top
chmod -R a-w work
```

The library is now ready for simulation by multiple simultaneous users, as follows:

```
vsim top -sv_lib test
```

The work/_dpi/exportwrapper argument provides a basename for the shared object.

At runtime, **vsim** automatically checks to see if the file *work/_dpi/exportwrapper.so* is up-to-date with respect to its C source code. If it is out of date, an error message is issued and elaboration stops.

DPI Arguments of Parameterized Datatypes

DPI import and export TF's can be written with arguments of parameterized data types. For example, assuming T1 and T2 are type parameters:

```
import "DPI-C" function T1 impf(input T2 arg);
```

This feature is only supported when **vopt** flow is used (VoptFlow = 1 in the *modelsim.ini* file). On occasion, the tool may not be able to resolve the type parameters while building the optimized design, in which case the workaround is to rewrite the function without using parameterized types. That the LRM rules for tf signature matching apply to the finally resolved value of type parameters. See P1800-2005 SystemVerilog LRM, Section 26.4.4 for further information on matching rules.

Allowing Verilog Functions to be Called from C

Certain C applications do not lend themselves easily to being called from context import TF’s, such as multithreaded C testbenches, or complex 3rd party integration applications. If you need to interact with Verilog from C models, an effective way is through using the tool’s “DPI-out-of-the-blue” functionality. Only calls to functions are allowed, tasks are not.

You can set the tool to allow “out-of-the-blue” Verilog function calls either for all simulations (DpiOutOfTheBlue = 1 in *modelsim.ini* file), or for a specific simulation (vsim -dpioutoftheblue 1).

Compiling and Linking C Applications for PLI/VPI/DPI

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C applications so that they can be loaded by ModelSim. Various native C/C++ compilers are supported on different platforms. The gcc compiler is supported on all platforms.

The following PLI/VPI/DPI routines are declared in the include files located in the ModelSim *<install_dir>/modeltech/include* directory:

- acc_user.h — declares the ACC routines
- veriuser.h — declares the TF routines
- vpi_user.h — declares the VPI routines
- svdpi.h — declares DPI routines

The following instructions assume that the PLI, VPI, or DPI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instructions.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries for PLI/VPI see PLI/VPI file loading. For DPI loading instructions, see DPI File Loading.

For all UNIX Platforms

The information in this section applies to all UNIX platforms.

app.so

If *app.so* is not in your current directory, you must tell the OS where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the command line option or control variable (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:

  LD_LIBRARY_PATH= <*library path without filename*> (for Solaris/Linux)

  or

  SHLIB_PATH= <*library path without filename*> (for HP-UX)

Windows Platforms

- Microsoft Visual C 4.1 or Later

  ```
  cl -c -I<install_dir>\modeltech\include app.c
  link -dll -export:<init_function> app.obj <install_dir>\win32\mtipli.lib -out:app.dll
  ```

  For the Verilog PLI, the <init_function> should be "init_usertfs". Alternatively, if there is no init_usertfs function, the <init_function> specified on the command line should be "veriusertfs". For the Verilog VPI, the <init_function> should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

If you need to run the profiler (see Profiling Performance and Memory Use) on a design that contains PLI/VPI code, add these two switches to the link commands shown above:

```
/DEBUG /DEBUGTYPE:COFF
```

These switches add symbols to the *.dll* that the profiler can use in its report.

- MinGW gcc 3.2.3

```
gcc -c -I<install_dir>\include app.c
gcc -shared -o app.dll app.o -L<install_dir>\win32 -lmtipli
```

ModelSim requires the use of MinGW gcc compiler rather than the Cygwin gcc compiler. MinGW gcc is available on the ModelSim FTP site. Remember to add the path to your gcc executable in the Windows environment variables.

DPI Imports on Windows Platforms

When linking the shared objects, be sure to specify one export option for each DPI imported task or function in your linking command line. You can use the **-isymfile** argument from the vlog command to obtain a complete list of all imported tasks/functions expected by ModelSim.

As an alternative to specifying one -export option for each imported task or function, you can make use of the __declspec (dllexport) macro supported by Visual C. You can place this macro before every DPI import task or function declaration in your C source. All the marked functions will be available for use by **vsim** as DPI import tasks and functions.

DPI Flow for Exported Tasks and Functions on Windows Platforms

Since the Windows platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates an object file <objname>.obj that contains "glue" code for exported tasks and functions. You must add that object file to the link line for your *.dll*, listed after the other object files. For example, a link line for MinGW would be:

```
gcc -shared -o app.dll app.obj <objname>.obj
        -L<install_dir>\modeltech\win32 -lmtipli
```

and a link line for Visual C would be:

```
link -dll -export:<init_function> app.obj <objname>.obj\
        <install_dir>\modeltech\win32\mtipli.lib -out:app.dll
```

32-bit Linux Platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- gcc compiler

  ```
  gcc -c -I/<install_dir>/modeltech/include app.c
  ld -shared -E -Bsymbolic -o app.so app.o -lc
  ```

  When using -Bsymbolic with ld, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored. If you are using ModelSim RedHat version 7.1 or below, you also need to add the -noinhibit-exec switch when you specify -Bsymbolic.

  The compiler switch -freg-struct-return must be used when compiling any FLI application code that contains foreign functions that return real or time values.

64-bit Linux for IA64 Platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- gcc compiler (gcc 3.2 or later)

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
  ```

  If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
  ```

64-bit Linux for Opteron/Athlon 64 and EM64T Platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- gcc compiler (gcc 3.2 or later)

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
  ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
  ```

  To compile for 32-bit operation, specify the -m32 argument on the gcc command line.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

```
gcc -c -fPIC -I/<install_dir>/modeltech/include math_app.c
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris Platform

If your PLI/VPI/DPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

- gcc compiler

  ```
  gcc -c -I/<install_dir>/modeltech/include app.c
  ld -G -Bsymbolic -o app.so app.o -lc
  ```

- cc compiler

  ```
  cc -c -I/<install_dir>/modeltech/include app.c
  ld -G -Bsymbolic -o app.so app.o -lc
  ```

When using **-Bsymbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

64-bit Solaris Platform

- gcc compiler

  ```
  gcc -c -I<install_dir>/modeltech/include -m64 -fPIC app.c
  gcc -shared -o app.so -m64 app.o
  ```

  This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH environment variable.

- cc compiler

  ```
  cc -v -xarch=v9 -O -I<install_dir>/modeltech/include -c app.c
  ld -G -Bsymbolic app.o -o app.so
  ```

When using **-Bsymbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

32-bit HP700 Platform

A shared library is created by creating object files that contain position-independent code (use the **+z** or **-fPIC** compiler argument) and by linking as a shared library (use the **-b** linker argument).

If your PLI/VPI/DPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command.

- gcc compiler

  ```
  gcc -c -fPIC -I/<install_dir>/modeltech/include app.c
  ld -b -o app.sl app.o -lc
  ```

  Note that **-fPIC** may not work with all versions of gcc.

- cc compiler

  ```
  cc -c +z +DD32 -I/<install_dir>/modeltech/include app.c
  ld -b -o app.sl app.o -lc
  ```

64-bit HP Platform

- cc compiler

  ```
  cc -v +DD64 -O -I<install_dir>/modeltech/include -c app.c
  ld -b -o app.sl app.o -lc
  ```

64-bit HP for IA64 Platform

- cc compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

  ```
  cc -c +DD64 -I/<install_dir>/modeltech/include app.c
  ld -b -o app.sl app.o
  ```

  If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

  ```
  cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
  ld -b -o math_app.sl math_app.o -lm
  ```

32-bit IBM RS/6000 Platform

ModelSim loads shared libraries on the IBM RS/6000 workstation. The shared library must import ModelSim's PLI/VPI/DPI symbols, and it must export the PLI or VPI application's initialization function or table. The ModelSim tool's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your PLI/VPI/DPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI/DPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command. The resulting object must be marked as shared reentrant using these **gcc** or **cc** compiler commands for AIX 4.x:

- gcc compiler

```
gcc -c -I/<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

- cc compiler

```
cc -c -I/<install_dir>/modeltech/include app.c
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". For the VPI, the exported symbol should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the shared object.

DPI Imports on 32-bit IBM RS/6000 Platform

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Flow for Exported Tasks and Functions on 32-bit IBM RS/6000 Platform

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

64-bit IBM RS/6000 Platform

Only versions 5.1 and later of AIX support the 64-bit platform. A gcc 64-bit compiler is not available at this time.

- VisualAge cc compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.c
ld -o app.s1 app.o -b64 -bE:app.exports \
    -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

DPI Imports on 64-bit IBM RS/6000 Platform

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Flow for Exported Tasks and Functions on 64-bit IBM RS/6000 Platform

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
        -bnoentry -lc
```

Compiling and Linking C++ Applications for PLI/VPI/DPI

ModelSim does not have direct support for any language other than standard C; however, C++ code can be loaded and executed under certain conditions.

Since ModelSim's PLI/VPI/DPI functions have a standard C prototype, you must prevent the C++ compiler from mangling the PLI/VPI/DPI function names. This can be accomplished by using the following type of extern:

```
extern "C"
{
  <PLI/VPI/DPI application function prototypes>
}
```

The header files *veriuser.h*, *acc_user.h*, and *vpi_user.h*, *svdpi.h*, and *dpiheader.h* already include this type of extern. You must also put the PLI/VPI/DPI shared library entry point (veriusertfs, init_usertfs, or vlog_startup_routines) inside of this type of extern.

The following platform-specific instructions show you how to compile and link your PLI/VPI/DPI C++ applications so that they can be loaded by ModelSim.

Although compilation and simulation switches are platform-specific, loading shared libraries is the same for all platforms. For information on loading libraries, see DPI File Loading.

For PLI/VPI only

If *app.so* is not in your current directory you must tell Solaris where to search for the shared object. You can do this one of two ways:

- Add a path before *app.so* in the foreign attribute specification. (The path may include environment variables.)
- Put the path in a UNIX shell environment variable:
  LD_LIBRARY_PATH= <*library path without filename*>

Windows Platforms

- Microsoft Visual C++ 4.1 or Later

  ```
  cl -c [-GX] -I<install_dir>\modeltech\include app.cxx
  link -dll -export:<init_function> app.obj
                                    <install_dir>\modeltech\win32\mtipli.lib /out:app.dll
  ```

  The **-GX** argument enables exception handling.

  For the Verilog PLI, the **<init_function>** should be "init_usertfs". Alternatively, if there is no init_usertfs function, the **<init_function>** specified on the command line should be "veriusertfs". For the Verilog VPI, the **<init_function>** should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the DLL.

  When executing **cl** commands in a DO file, use the **/NOLOGO** switch to prevent the Microsoft C compiler from writing the logo banner to stderr. Writing the logo causes Tcl to think an error occurred.

  If you need to run the profiler (see Profiling Performance and Memory Use) on a design that contains PLI/VPI code, add these two switches to the link command shown above:

  ```
  /DEBUG /DEBUGTYPE:COFF
  ```

  These switches add symbols to the *.dll* that the profiler can use in its report.

- MinGW C++ Version 3.2.3

  ```
  g++ -c -I<install_dir>\modeltech\include app.cpp
  g++ -shared -o app.dll app.o -L<install_dir>\modeltech\win32 -lmtipli
  ```

  ModelSim requires the use of MinGW gcc compiler rather than the Cygwin gcc compiler.

DPI Imports on Windows Platforms

When linking the shared objects, be sure to specify one -export option for each DPI imported task or function in your linking command line. You can use Verilog's **-isymfile** option to obtain a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the Windows platform lacks the necessary runtime linking complexity, you must perform an additional manual step in order to compile the HDL source files into the shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The -dpiexportobj generates the object file <objname>.obj that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, if the object name was *dpi1*, the link line for MinGW would be:

```
g++ -shared -o app.dll app.obj <objname>.obj
        -L<install_dir>\modeltech\win32 -lmtipli
```

32-bit Linux Platform

- GNU C++ Version 2.95.3 or Later

```
g++ -c -fPIC -I<install_dir>/modeltech/include app.cpp
g++ -shared -fPIC -o app.so app.o
```

64-bit Linux for IA64 Platform

64-bit Linux is supported on RedHat Linux Advanced Workstation 2.1 for Itanium 2.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library libm, specify '-lm' to the 'ld' command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

64-bit Linux for Opteron/Athlon 64 and EM64T Platforms

64-bit Linux is supported on RedHat Linux EWS 3.0 for Opteron/Athlon 64 and EM64T.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o app.so app.o
```

To compile for 32-bit operation, specify the -m32 argument on the gcc command line.

If your PLI/VPI/DPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI/DPI application. For example, to use the system math library libm, specify -lm to the ld command:

```
g++ -c -fPIC -I/<install_dir>/modeltech/include math_app.cpp
ld -shared -Bsymbolic -E --allow-shlib-undefined -o math_app.so math_app.o -lm
```

32-bit Solaris Platform

If your PLI/VPI application uses anything from a system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```

- Sun Forte C++ Compiler

```
cc -c -I/<install_dir>/modeltech/include app.cpp
ld -G -Bsymbolic -o app.so app.o -lc
```

When using **-Bsymbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

64-bit Solaris Platform

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I<install_dir>/modeltech/include -m64 -fPIC app.cpp
g++ -shared -o app.so -m64 app.o
```

This was tested with gcc 3.2.2. You may need to add the location of *libgcc_s.so.1* to the LD_LIBRARY_PATH environment variable.

- cc compiler

```
cc -v -xarch=v9 -O -I<install_dir>/modeltech/include -c app.cpp
ld -G -Bsymbolic app.o -o app.so
```

When using **-Bsymbolic** with **ld**, all symbols are first resolved within the shared library at link time. This will result in a list of undefined symbols. This is only a warning for shared libraries and can be ignored.

32-bit HP700 Platform

A shared library is created by creating object files that contain position-independent code (use the **+z** or **-fPIC** compiler argument) and by linking as a shared library (use the **-b** linker argument).

If your PLI/VPI application uses anything from a system library, you’ll need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify ‘-lc’ to the ‘ld’ command.

- GNU C++ compiler

```
g++ -c -fPIC -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

- cc compiler

```
cc -c +z +DD32 -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o -lc
```

Note that **-fPIC** may not work with all versions of gcc.

64-bit HP Platform

- cc Compiler

```
cc -v +DD64 -O -I<install_dir>/modeltech/include -c app.cpp
ld -b -o app.sl app.o -lc
```

64-bit HP for IA64 Platform

- HP ANSI C++ Compiler (/opt/ansic/bin/cc, /usr/ccs/bin/ld)

```
cc -c +DD64 -I/<install_dir>/modeltech/include app.cpp
ld -b -o app.sl app.o
```

If your PLI/VPI application requires a user or vendor-supplied C library, or an additional system library, you will need to specify that library when you link your PLI/VPI application. For example, to use the system math library, specify '-lm' to the 'ld' command:

```
cc -c +DD64 -I/<install_dir>/modeltech/include math_app.c
ld -b -o math_app.sl math_app.o -lm
```

32-bit IBM RS/6000 Platform

ModelSim loads shared libraries on the IBM RS/6000 workstation. The shared library must import ModelSim's PLI/VPI symbols, and it must export the PLI or VPI application’s initialization function or table. The ModelSim tool's export file is located in the ModelSim installation directory in *rs6000/mti_exports*.

If your PLI/VPI application uses anything from a system library, you'll need to specify that library when you link your PLI/VPI application. For example, to use the standard C library, specify '-lc' to the 'ld' command. The resulting object must be marked as shared reentrant using these **gcc** or **cc** compiler commands for AIX 4.x:

- GNU C++ compiler version gcc 3.2 or later

```
g++ -c -I/<install_dir>/modeltech/include app.cpp
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

- VisualAge C++ Compiler

```
cc -c -I/<install_dir>/modeltech/include app.cpp
ld -o app.sl app.o -bE:app.exp \
    -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE -bnoentry -lc
```

The *app.exp* file must export the PLI/VPI initialization function or table. For the PLI, the exported symbol should be "init_usertfs". Alternatively, if there is no init_usertfs function, then the exported symbol should be "veriusertfs". For the VPI, the exported symbol should be "vlog_startup_routines". These requirements ensure that the appropriate symbol is exported, and thus ModelSim can find the symbol when it dynamically loads the shared object.

For DPI Imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.dll app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
        -bnoentry -lc
```

64-bit IBM RS/6000 Platform

Only version 5.1 and later of AIX supports the 64-bit platform. A gcc 64-bit compiler is not available at this time.

- VisualAge C++ Compiler

```
cc -c -q64 -I/<install_dir>/modeltech/include app.cpp
ld -o app.s1 app.o -b64 -bE:app.exports \
    -bI:/<install_dir>/modeltech/rs64/mti_exports -bM:SRE -bnoentry -lc
```

For DPI Imports

When linking the shared objects, be sure to specify **-bE:<isymfile>** option on the link command line. <isymfile> is the name of the file generated by the**-isymfile** argument to the vlog command. Once you have created the <isymfile>, it contains a complete list of all imported tasks and functions expected by ModelSim.

DPI Special Flow for Exported Tasks and Functions

Since the RS6000 platform lacks the necessary runtime linking capabilities, you must perform an additional manual step in order to prepare shared objects containing calls to exported SystemVerilog tasks or functions shared object file. You need to invoke a special run of vsim. The command is as follows:

```
vsim <top du list> -dpiexportobj <objname> <other args>
```

The **-dpiexportobj** generates the object file <objname>.o that contains "glue" code for exported tasks and functions. You must add that object file to the link line, listed after the other object files. For example, a link line would be:

```
ld -o app.so app.o <objname>.o
        -bE:<isymfile> -bI:/<install_dir>/modeltech/rs6000/mti_exports -bM:SRE
        -bnoentry -lc
```

Specifying Application Files to Load

PLI and VPI file loading is identical. DPI file loading uses switches to the **vsim** command.

PLI/VPI file loading

The PLI/VPI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a **-pli** argument to the simulator (multiple arguments are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

Note

On Windows platforms, the file names shown above should end with *.dll* rather than *.so*.

The various methods of specifying PLI/VPI applications can be used simultaneously. The libraries are loaded in the order listed above. Environment variable references can be used in the paths to the libraries in all cases.

See also Simulator Variables for more information on the *modelsim.ini* file.

DPI File Loading

DPI applications are specified to vsim using the following SystemVerilog arguments:

Table D-1.

argument	description
-sv_lib <name>	specifies a library name to be searched and used. No filename extensions must be specified. (The extensions ModelSim expects are: *.sl* for HP, *.dll* for Win32, *.so* for all other platforms.)
-sv_root <name>	specifies a new prefix for shared objects as specified by -sv_lib
-sv_liblist	specifies a “bootstrap file” to use

When the simulator finds an imported task or function, it searches for the symbol in the collection of shared objects specified using these arguments.

For example, you can specify the DPI application as follows:

```
vsim -sv_lib dpiapp1 -sv_lib dpiapp2 -sv_lib dpiappn
```

It is a mistake to specify DPI import tasks and functions (tf) inside PLI/VPI shared objects. However, a DPI import tf can make calls to PLI/VPI C code, providing that **vsim -gblso** was used to mark the PLI/VPI shared object with global symbol visibility. See Loading Shared Objects with Global Symbol Visibility.

Loading Shared Objects with Global Symbol Visibility

On Unix platforms you can load shared objects such that all symbols in the object have global visibility. To do this, use the **-gblso** argument to **vsim** when you load your PLI/VPI application. For example:

vsim -pli obj1.so -pli obj2.so -gblso obj1.so

The **-gblso** argument works in conjunction with the GlobalSharedObjectList variable in the *modelsim.ini* file. This variable allows user C code in other shared objects to refer to symbols in a shared object that has been marked as global. All shared objects marked as global are loaded by the simulator earlier than any non-global shared objects.

PLI Example

The following example is a trivial, but complete PLI application.

```
hello.c:
   #include "veriuser.h"
   static PLI_INT32 hello()
   {
      io_printf("Hi there\n");
      return 0;
   }
   s_tfcell veriusertfs[] = {
      {usertask, 0, 0, 0, hello, 0, "$hello"},
      {0}  /* last entry must be 0 */
   };
hello.v:
   module hello;
      initial $hello;
   endmodule
Compile the PLI code for the Solaris operating system:
   % cc -c -I<install_dir>/modeltech/include hello.c
   % ld -G -o hello.sl hello.o
Compile the Verilog code:
   % vlib work
   % vlog hello.v
Simulate the design:
   % vsim -c -pli hello.sl hello
   # Loading work.hello
   # Loading ./hello.sl
   VSIM 1> run -all
   # Hi there
   VSIM 2> quit
```

VPI Example

The following example is a trivial, but complete VPI application. A general VPI example can be found in *<install_dir>/modeltech/examples/verilog/vpi*.

hello.c:

```
   #include "vpi_user.h"
   static PLI_INT32 hello(PLI_BYTE8 * param)
   {
      vpi_printf( "Hello world!\n" );
      return 0;
   }
   void RegisterMyTfs( void )
   {
      s_vpi_systf_data systf_data;
      vpiHandle systf_handle;
      systf_data.type        = vpiSysTask;
      systf_data.sysfunctype = vpiSysTask;
      systf_data.tfname      = "$hello";
      systf_data.calltf      = hello;
      systf_data.compiletf   = 0;
      systf_data.sizetf      = 0;
      systf_data.user_data   = 0;
      systf_handle = vpi_register_systf( &systf_data );
      vpi_free_object( systf_handle );
   }
   void (*vlog_startup_routines[])() = {
      RegisterMyTfs,
      0
   };
hello.v:
   module hello;
      initial $hello;
   endmodule
Compile the VPI code for the Solaris operating system:
   % gcc -c -I<install_dir>/include hello.c
   % ld -G -o hello.sl hello.o
Compile the Verilog code:
   % vlib work
   % vlog hello.v
Simulate the design:
   % vsim -c -pli hello.sl hello
   # Loading work.hello
   # Loading ./hello.sl
   VSIM 1> run -all
   # Hello world!
   VSIM 2> quit
```

DPI Example

The following example is a trivial but complete DPI application. For win32 and RS6000 platforms, an additional step is required. For additional examples, see the *<install_dir>/modeltech/examples/systemverilog/dpi* directory.

```
hello_c.c:
#include "svdpi.h"
#include "dpiheader.h"
int c_task(int i, int *o)
{
   printf("Hello from c_task()\n");
   verilog_task(i, o); /* Call back into Verilog */
   *o = i;
   return(0); /* Return success (required by tasks) */
}
hello.v:
module hello_top;
   int ret;
   export "DPI-C" task verilog_task;
   task verilog_task(input int i, output int o);
      #10;
      $display("Hello from verilog_task()");
   endtask
   import "DPI-C" context task c_task(input int i, output int o);
   initial
   begin
      c_task(1, ret);  // Call the c task named 'c_task()'
   end
endmodule
Compile the Verilog code:
   % vlib work
   % vlog -sv -dpiheader dpiheader.h hello.v
Compile the DPI code for the Solaris operating system:
   % gcc -c -g -I<install_dir>/modeltech/include hello_c.c
   % ld -G -o hello_c.so hello_c.o
Simulate the design:
   % vsim -c -sv_lib hello_c hello_top
   # Loading work.hello_c
   # Loading ./hello_c.so
   VSIM 1> run -all
   # Hello from c_task()
   # Hello from verilog_task()
   VSIM 2> quit
```

The PLI Callback reason Argument

The second argument to a PLI callback function is the reason argument. The values of the various reason constants are defined in the *veriuser.h* include file. See IEEE Std 1364 for a description of the reason constants. The following details relate to ModelSim Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the misctf callback functions under the following circumstances:

`reason_endofcompile`
For the completion of loading the design.

`reason_finish`
For the execution of the $finish system task or the **quit** command.

`reason_startofsave`

For the start of execution of the **checkpoint** command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to tf_write_save() until it is called with reason_save.

`reason_save`

For the execution of the **checkpoint** command. This is when the PLI application must save its state with calls to tf_write_save().

`reason_startofrestart`

For the start of execution of the **restore** command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its state with calls to tf_read_restart() until it is called with reason_restart. The reason_startofrestart value is passed only for a restore command, and not in the case that the simulator is invoked with -restore.

`reason_restart`

For the execution of the **restore** command. This is when the PLI application must restore its state with calls to tf_read_restart().

`reason_reset`

For the execution of the **restart** command. This is when the PLI application should free its memory and reset its state. We recommend that all PLI applications reset their internal state during a restart as the shared library containing the PLI code might not be reloaded. (See the **-keeploaded** and **-keeploadedrestart** arguments to **vsim** for related information.)

`reason_endofreset`

For the completion of the **restart** command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`

For the execution of the $stop system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`

For the execution of the **environment** command or selecting a scope in the structure window. Also for the call to acc_set_interactive_scope() if the callback_flag argument is non-zero.

`reason_paramvc`

For the change of value on the system task or function argument.

`reason_synch`

For the end of time step event scheduled by tf_synchronize().

`reason_rosynch`

For the end of time step event scheduled by tf_rosynchronize().

`reason_reactivate`

For the simulation event scheduled by tf_setdelay().

`reason_paramdrc`

Not supported in ModelSim Verilog.

`reason_force`
Not supported in ModelSim Verilog.

`reason_release`
Not supported in ModelSim Verilog.

`reason_disable`
Not supported in ModelSim Verilog.

The sizetf Callback Function

A user-defined system function specifies the width of its return value with the sizetf callback function, and the simulator calls this function while loading the design. The following details on the sizetf callback function are not found in the IEEE Std 1364:

- If you omit the sizetf function, then a return width of 32 is assumed.
- The sizetf function should return 0 if the system function return value is of Verilog type "real".
- The sizetf function should return -32 if the system function return value is of Verilog type "integer".

PLI Object Handles

Many of the object handles returned by the PLI ACC routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the acc_close() routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after acc_close() is called. The following object types are created on demand in ModelSim Verilog:

accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)

If your PLI application uses these types of objects, then it is important to call acc_close() to free the memory allocated for these objects when the application is done using them.

If your PLI application places value change callbacks on accRegBit or accTerminal objects, *do not* call acc_close() while these callbacks are in effect.

Third Party PLI Applications

Many third party PLI applications come with instructions on using them with ModelSim Verilog. Even without the instructions, it is still likely that you can get it to work with

ModelSim Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with ModelSim Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a veriuser.c file. The veriuser.c file contains the registration information as described above in Registering PLI Applications. To prepare the application for ModelSim Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see Compiling and Linking C Applications for PLI/VPI/DPI). For example, if you have a *veriuser.c* file and a library archive *libapp.a* file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c
% ld -G -o app.sl veriuser.o libapp.a
```

The PLI application is now ready to be run with ModelSim Verilog. All that's left is to specify the resulting object file to the simulator for loading using the **Veriuser** entry in the *modesim.ini* file, the **-pli** simulator argument, or the PLIOBJS environment variable (see Registering PLI Applications).

Note

On the HP700 platform, the object files must be compiled as position-independent code by using the **+z** compiler argument. Since, the object files supplied for Verilog-XL may be compiled for static linking, you may not be able to use the object files to create a dynamically loadable object for ModelSim Verilog. In this case, you must get the third party application vendor to supply the object files compiled as position-independent code.

Support for VHDL Objects

The PLI ACC routines also provide limited support for VHDL objects in either an all VHDL design or a mixed VHDL/Verilog design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

Table D-2.

Type	Fulltype	Description
accArchitecture	accArchitecture	instantiation of an architecture
accArchitecture	accEntityVitalLevel0	instantiation of an architecture whose entity is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel0	instantiation of an architecture which is marked with the attribute VITAL_Level0
accArchitecture	accArchVitalLevel1	instantiation of an architecture which is marked with the attribute VITAL_Level1

Table D-2.

Type	Fulltype	Description
accArchitecture	accForeignArch	instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics
accArchitecture	accForeignArchMixed	instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics
accBlock	accBlock	block statement
accForLoop	accForLoop	for loop statement
accForeign	accShadow	foreign scope created by mti_CreateRegion()
accGenerate	accGenerate	generate statement
accPackage	accPackage	package declaration
accSignal	accSignal	signal declaration

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, variables, attributes, subprograms, and processes. However, some of these objects can be manipulated through the ModelSim VHDL foreign interface (mti_* routines). See the *FLI Reference Manual* for more information.

IEEE Std 1364 ACC Routines

ModelSim Verilog supports the following ACC routines:

Table D-3.

Routines
acc_append_delays
acc_append_pulsere
acc_close
acc_collect
acc_compare_handles
acc_configure
acc_count
acc_fetch_argc
acc_fetch_argv
acc_fetch_attribute
acc_fetch_attribute_int
acc_fetch_attribute_str
acc_fetch_defname
acc_fetch_delay_mode
acc_fetch_delays
acc_fetch_direction
acc_fetch_edge
acc_fetch_fullname
acc_fetch_fulltype
acc_fetch_index
acc_fetch_location
acc_fetch_name
acc_fetch_paramtype
acc_fetch_paramval
acc_fetch_polarity
acc_fetch_precision
acc_fetch_pulsere
acc_fetch_range
acc_fetch_size
acc_fetch_tfarg
acc_fetch_itfarg
acc_fetch_tfarg_int
acc_fetch_itfarg_int
acc_fetch_tfarg_str
acc_fetch_itfarg_str
acc_fetch_timescale_info
acc_fetch_type
acc_fetch_type_str
acc_fetch_value
acc_free
acc_handle_by_name
acc_handle_calling_mod_m
acc_handle_condition
acc_handle_conn
acc_handle_hiconn
acc_handle_interactive_scope
acc_handle_loconn
acc_handle_modpath
acc_handle_notifier
acc_handle_object
acc_handle_parent
acc_handle_path
acc_handle_pathin
acc_handle_pathout
acc_handle_port
acc_handle_scope
acc_handle_simulated_net
acc_handle_tchk
acc_handle_tchkarg1
acc_handle_tchkarg2
acc_handle_terminal
acc_handle_tfarg
acc_handle_itfarg
acc_handle_tfinst
acc_initialize
acc_next
acc_next_bit
acc_next_cell
acc_next_cell_load
acc_next_child
acc_next_driver
acc_next_hiconn
acc_next_input
acc_next_load
acc_next_loconn
acc_next_modpath
acc_next_net
acc_next_output
acc_next_parameter
acc_next_port
acc_next_portout
acc_next_primitive
acc_next_scope
acc_next_specparam
acc_next_tchk
acc_next_terminal
acc_next_topmod
acc_object_in_typelist
acc_object_of_type
acc_product_type
acc_product_version
acc_release_object
acc_replace_delays
acc_replace_pulsere
acc_reset_buffer
acc_set_interactive_scope
acc_set_pulsere
acc_set_scope
acc_set_value
acc_vcl_add
acc_vcl_delete
acc_version

acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

IEEE Std 1364 TF Routines

ModelSim Verilog supports the following TF (task and function) routines;

Table D-4.

Routines
io_mcdprintf
io_printf
mc_scan_plusargs
tf_add_long
tf_asynchoff
tf_iasynchoff
tf_asynchon
tf_iasynchon
tf_clearalldelays
tf_iclearalldelays
tf_compare_long
tf_copypvc_flag
tf_icopypvc_flag
tf_divide_long
tf_dofinish
tf_dostop
tf_error
tf_evaluatep
tf_ievaluatep
tf_exprinfo
tf_iexprinfo
tf_getcstringp
tf_igetcstringp
tf_getinstance
tf_getlongp
tf_igetlongp
tf_getlongtime
tf_igetlongtime
tf_getnextlongtime
tf_getp
tf_igetp
tf_getpchange
tf_igetpchange
tf_getrealp
tf_igetrealp
tf_getrealtime
tf_igetrealtime
tf_gettime
tf_igettime
tf_gettimeprecision
tf_igettimeprecision
tf_gettimeunit
tf_igettimeunit
tf_getworkarea
tf_igetworkarea
tf_long_to_real
tf_longtime_tostr
tf_message
tf_mipname
tf_imipname
tf_movepvc_flag
tf_imovepvc_flag
tf_multiply_long
tf_nodeinfo
tf_inodeinfo
tf_nump
tf_inump
tf_propagatep
tf_ipropagatep
tf_putlongp
tf_iputlongp
tf_putp
tf_iputp
tf_putrealp
tf_iputrealp
tf_read_restart
tf_real_to_long
tf_rosynchronize
tf_irosynchronize
tf_scale_longdelay
tf_scale_realdelay
tf_setdelay
tf_isetdelay
tf_setlongdelay
tf_isetlongdelay
tf_setrealdelay
tf_isetrealdelay
tf_setworkarea
tf_isetworkarea
tf_sizep
tf_isizep
tf_spname
tf_ispname
tf_strdelputp
tf_istrdelputp
tf_strgetp
tf_istrgetp
tf_strgettime
tf_strlongdelputp
tf_istrlongdelputp
tf_strrealdelputp
tf_istrrealdelputp
tf_subtract_long
tf_synchronize
tf_isynchronize
tf_testpvc_flag
tf_itestpvc_flag
tf_text
tf_typep
tf_itypep
tf_unscale_longdelay
tf_unscale_realdelay
tf_warning
tf_write_save

SystemVerilog DPI Access Routines

ModelSim SystemVerilog supports all routines defined in the "svdpi.h" file defined in P1800-2005, with the exception of functions related to open array processing.

Verilog-XL Compatible Routines

The following PLI routines are not defined in IEEE Std 1364, but ModelSim Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The condition argument must be a handle obtained from the acc_handle_condition routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the aof_hightime argument.

64-bit Support for PLI

The PLI function acc_fetch_paramval() cannot be used on 64-bit platforms to fetch a string value of a parameter. Because of this, the function acc_fetch_paramval_str() has been added to the PLI for this use. acc_fetch_paramval_str() is declared in acc_user.h. It functions in a manner similar to acc_fetch_paramval() except that it returns a char *. acc_fetch_paramval_str() can be used on all platforms.

Using 64-bit ModelSim with 32-bit Applications

If you have 32-bit PLI/VPI/DPI applications and wish to use 64-bit ModelSim, you will need to port your code to 64 bits by moving from the ILP32 data model to the LP64 data model. We strongly recommend that you consult the 64-bit porting guides for Sun and HP.

PLI/VPI Tracing

The foreign interface tracing feature is available for tracing PLI and VPI function calls. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files that can be used to replay what the foreign interface code did.

The Purpose of Tracing Files

The purpose of the logfile is to aid you in debugging PLI or VPI code. The primary purpose of the replay facility is to send the replay files to MTI support for debugging co-simulation problems, or debugging PLI/VPI problems for which it is impractical to send the PLI/VPI code. We still need you to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

Invoking a Trace

To invoke the trace, call vsim with the **-trace_foreign** argument:

Syntax

```
vsim
```
-trace_foreign <action> [-tag <name>]

Arguments

```
<action>
```
Specifies one of the following actions:

Table D-5.

Value	Action	Result
1	create log only	writes a local file called "mti_trace_<tag>"
2	create replay only	writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c"
3	create both log and replay	

```
-tag <name>
```
Used to give distinct file names for multiple traces. Optional.

Examples

```
vsim -trace_foreign 1 mydesign
```
Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```
Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```
Creates a logfile with a tag of "2".

The tracing operations will provide tracing during all user foreign code-calls, including PLI/VPI user tasks and functions (calltf, checktf, sizetf and misctf routines), and Verilog VCL callbacks.

Checkpointing and PLI/VPI/DPI Code

The checkpoint feature in ModelSim captures the state of PLI/VPI/DPI code. See The PLI Callback reason Argument for reason arguments that apply to checkpoint/restore.

Checkpointing Code that Works with Heap Memory

If checkpointing code that works with heap memory, use mti_Malloc() rather than raw malloc() or new. Any memory allocated with mti_Malloc() is guaranteed to be restored correctly. Any memory allocated with raw malloc() will not be restored correctly, and simulator crashes can result.

Debugging PLI/VPI/DPI Application Code

ModelSim offers the optional C Debug feature. This tool allows you to interactively debug SystemC/C/C++ source code with the open-source **gdb** debugger. See C Debug for details. If you don’t have access to C Debug, continue reading for instructions on how to attach to an external C debugger.

In order to debug your PLI/VPI/DPI application code in a debugger, you must first:

1. Compile the application code with debugging information (using the **-g** option) and without optimizations (for example, don’t use the **-O** option).
2. Load **vsim** into a debugger.

   Even though **vsim** is stripped, most debuggers will still execute it. You can invoke the debugger directly on **vsimk**, the simulation kernal where your application code is loaded (for example, "ddd `which vsimk`"), or you can attach the debugger to an already running **vsim** process. In the second case, you must attach to the PID for **vsimk**, and you must specify the full path to the **vsimk** executable (for example, "gdb $MTI_HOME/sunos5/vsimk 1234").

   On Solaris, AIX, and Linux systems you can use either **gdb** or **ddd**. On HP-UX systems you can use the **wdb** debugger from HP. You will need version 1.2 or later.
3. Set an entry point using breakpoint.

   Since initially the debugger recognizes only **vsim's** PLI/VPI/DPI function symbols, when invoking the debugger directly on **vsim** you need to place a breakpoint in the first PLI/VPI/DPI function that is called by your application code. An easy way to set an entry point is to put a call to acc_product_version() as the first executable statement in your application code. Then, after **vsim** has been loaded into the debugger, set a breakpoint in this function. Once you have set the breakpoint, run **vsim** with the usual arguments.

When the breakpoint is reached, the shared library containing your application code has been loaded.

4. In some debuggers, you must use the **share** command to load the application's symbols.

At this point all of the application's symbols should be visible. You can now set breakpoints in and single step through your application code.

Troubleshooting a Missing DPI Import Function

DPI uses C function linkage. If your DPI application is written in C++, it is important to remember to use extern "C" declaration syntax appropriately. Otherwise the C++ compiler will produce a mangled C++ name for the function, and the simulator is not able to locate and bind the DPI call to that function.

HP-UX Specific Warnings

On HP-UX you might see some warning messages that **vsim** does not have debugging information available. This is normal. If you are using Exceed to access an HP machine from Windows NT, it is recommended that you run **vsim** in command line or batch mode because your NT machine may hang if you run **vsim** in GUI mode. Click on the "go" button, or use F5 or the **go** command to execute **vsim** in **wdb**.

You might also see a warning about not finding "__dld_flags" in the object file. This warning can be ignored. You should see a list of libraries loaded into the debugger. It should include the library for your PLI/VPI/DPI application. Alternatively, you can use **share** to load only a single library.

Appendix E
Command and Keyboard Shortcuts

This appendix is a collection of the keyboard and command shortcuts available in the ModelSim GUI.

Command Shortcuts

- You may abbreviate command syntax, but there’s a catch — the minimum number of characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work. For this reason ModelSim does not allow command name abbreviations in macro files. This minimizes your need to update macro files as new commands are added.
- Multiple commands may be entered on one line if they are separated by semi-colons (;). For example:

  ```
  ModelSim> vlog -nodebug=ports level3.v level2.v ; vlog -nodebug top.v
  ```

  The return value of the last function executed is the only one printed to the transcript. This may cause some unexpected behavior in certain circumstances. Consider this example:

  ```
  vsim -c -do "run 20 ; simstats ; quit -f" top
  ```

  You probably expect the **simstats** results to display in the Transcript window, but they will not, because the last command is **quit -f**. To see the return values of intermediate commands, you must explicitly print the results. For example:

  ```
  vsim -do "run 20 ; echo [simstats]; quit -f" -c top
  ```

Command History Shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the ModelSim/VSIM prompt:

Table E-1.

Shortcut	Description
!!	repeats the last command
!n	repeats command number n; n is the VSIM prompt number (e.g., for this prompt: VSIM 12>, n =12)
!abc	repeats the most recent command starting with "abc"

Table E-1.

Shortcut	Description
^xyz^ab^	replaces "xyz" in the last command with "ab"
up and down arrows	scrolls through the command history with the keyboard arrows
click on prompt	left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor
his or history	shows the last few commands (up to 50 are kept)

Main and Source Window Mouse and Keyboard Shortcuts

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all **Notepad** windows (enter the **notepad** command within ModelSim to open the Notepad editor).

Table E-2.

Mouse - UNIX	Mouse - Windows	Result
< left-button - click >		move the insertion cursor
< left-button - press > + drag		select
< shift - left-button - press >		extend selection
< left-button - double-click >		select word
< left-button - double-click > + drag		select word + word
< control - left-button - click >		move insertion cursor without changing the selection
< left-button - click > on previous ModelSim or VSIM prompt		copy and paste previous command string to current prompt
< middle-button - click >	none	paste clipboard
< middle-button - press > + drag	none	scroll the window

Table E-3.

Keystrokes - UNIX	Keystrokes - Windows	Result
< left | right arrow >		move cursor left | right one character
< control > < left | right arrow >		move cursor left | right one word
< shift > < left | right | up | down arrow >		extend selection of text
< control > < shift > < left | right arrow >		extend selection of text by word
< up | down arrow >		scroll through command history (in Source window, moves cursor one line up | down)
< control > < up | down >		moves cursor up | down one paragraph
< control > < home >		move cursor to the beginning of the text
< control > < end >		move cursor to the end of the text
< backspace >, < control-h >	< backspace >	delete character to the left
< delete >, < control-d >	< delete >	delete character to the right
none	esc	cancel
< alt >		activate or inactivate menu bar mode
< alt > < F4 >		close active window
< control - a >, < home >	< home >	move cursor to the beginning of the line
< control - b >		move cursor left
< control - d >		delete character to the right
< control - e >, < end >	< end >	move cursor to the end of the line
< control - f >	<right arrow>	move cursor right one character
< control - k >		delete to the end of line

Table E-3.

Keystrokes - UNIX	Keystrokes - Windows	Result
< control - n >		move cursor one line down (Source window only under Windows)
< control - o >	none	insert a new line character at the cursor
< control - p >		move cursor one line up (Source window only under Windows)
< control - s >	< control - f >	find
< F3 >		find next
< control - t >		reverse the order of the two characters on either side of the cursor
< control - u >		delete line
< control - v >, PageDn	PageDn	move cursor down one screen
< control - w >	< control - x >	cut the selection
< control - x >, < control - s>	< control - s >	save
< control - y >, F18	< control - v >	paste the selection
none	< control - a >	select the entire contents of the widget
< control - \ >		clear any selection in the widget
< control - ->, < control - / >	< control - Z >	undoes previous edits in the Source window
< meta - "<" >	none	move cursor to the beginning of the file
< meta - ">" >	none	move cursor to the end of the file
< meta - v >, PageUp	PageUp	move cursor up one screen
< Meta - w>	< control - c >	copy selection
< F8 >		search for the most recent command that matches the characters typed (Main window only)

Table E-3.

Keystrokes - UNIX	Keystrokes - Windows	Result
< F9>		run simulation
< F10 >		continue simulation
	< F11 >	single-step
	< F12>	step-over

The Main window allows insertions or pastes only after the prompt; therefore, you don't need to set the cursor when copying strings to the command line.

List Window Keyboard Shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

Table E-4.

Key	Action
<left arrow>	scroll listing left (selects and highlights the item to the left of the currently selected item)
<right arrow>	scroll listing right (selects and highlights the item to the right of the currently selected item)
<up arrow>	scroll listing up
<down arrow>	scroll listing down
<page up> <control-up arrow>	scroll listing up by page
<page down> <control-down arrow>	scroll listing down by page
<tab>	searches forward (down) to the next transition on the selected signal
<shift-tab>	searches backward (up) to the previous transition on the selected signal (does not function on HP workstations)
<shift-left arrow> <shift-right arrow>	extends selection left/right
<control-f> Windows <control-s> UNIX	opens the Find dialog box to find the specified item label within the list display

Wave Window Mouse and Keyboard Shortcuts

The following mouse actions and keystrokes can be used in the Wave window.

Table E-5.

Mouse action	Result
< control - left-button - drag down and right>[1]	zoom area (in)
< control - left-button - drag up and right>	zoom out
< control - left-button - drag up and left>	zoom fit
<left-button - drag> (Select mode) < middle-button - drag> (Zoom mode)	moves closest cursor
< control - left-button - click on a scroll arrow >	scrolls window to very top or bottom (vertical scroll) or far left or right (horizontal scroll)
< middle mouse-button - click in scroll bar trough> (UNIX) only	scrolls window to position of click

[1] If you enter zoom mode by selecting **View > Zoom > Mouse Mode > Zoom Mode**, you do not need to hold down the <Ctrl> key.

Table E-6.

Keystroke	Action
s	bring into view and center the currently active cursor
i I or +	zoom in (mouse pointer must be over the cursor or waveform panes)
o O or -	zoom out (mouse pointer must be over the cursor or waveform panes)
f or F	zoom full (mouse pointer must be over the cursor or waveform panes)
l or L	zoom last (mouse pointer must be over the cursor or waveform panes)
r or R	zoom range (mouse pointer must be over the cursor or waveform panes)
<up arrow>/ <down arrow>	with mouse over waveform pane, scrolls entire window up/down one line; with mouse over pathname or values pane, scrolls highlight up/down one line
<left arrow>	scroll pathname, values, or waveform pane left
<right arrow>	scroll pathname, values, or waveform pane right

Table E-6.

Keystroke	Action
<page up>	scroll waveform pane up by a page
<page down>	scroll waveform pane down by a page
<tab>	search forward (right) to the next transition on the selected signal - finds the next edge
<shift-tab>	search backward (left) to the previous transition on the selected signal - finds the previous edge
<control-f> Windows <control-s> UNIX	open the find dialog box; searches within the specified field in the pathname pane for text strings
<control-left arrow>	scroll pathname, values, or waveform pane left by a page
<control-right arrow>	scroll pathname, values, or waveform pane right by a page

Appendix F
Setting GUI Preferences

The ModelSim GUI is programmed using Tcl/Tk. It is highly customizable. You can control everything from window size, position, and color to the text of window prompts, default output filenames, and so forth. You can even add buttons and menus that run user-programmable Tcl code.

Most user GUI preferences are stored as Tcl variables in the *.modelsim* file on Unix/Linux platforms or the Registry on Windows platforms. The variable values save automatically when you exit ModelSim. Some of the variables are modified by actions you take with menus or windows (e.g., resizing a window changes its geometry variable). Or, you can edit the variables directly either from the ModelSim > prompt or the Edit Preferences dialog.

Customizing the Simulator GUI Layout

You can customize the layout of panes, windows, toolbars, etc. This section discusses layouts and how they are used in ModelSim.

Layouts and Modes of Operation

ModelSim ships with three default layouts that correspond to three modes of operation.

Table F-1.

Layout	**Mode**
NoDesign	a design is not yet loaded
Simulate	a design is loaded
Coverage	a design is loaded with code coverage enabled

As you load and unload designs, ModelSim switches between the layouts.

Custom Layouts

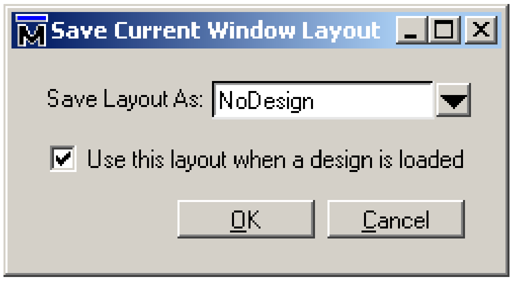
You can create custom layouts or modify the three default layouts.

Creating Custom Layouts

To create a custom layout or modify one of the default layouts, follow these steps:

1. Rearrange the GUI as you see fit (see Navigating the Graphic User Interface for details).

2. Select **Window** > **Layouts** > **Save**.



Save Layout As: NoDesign

☑ Use this layout when a design is loaded

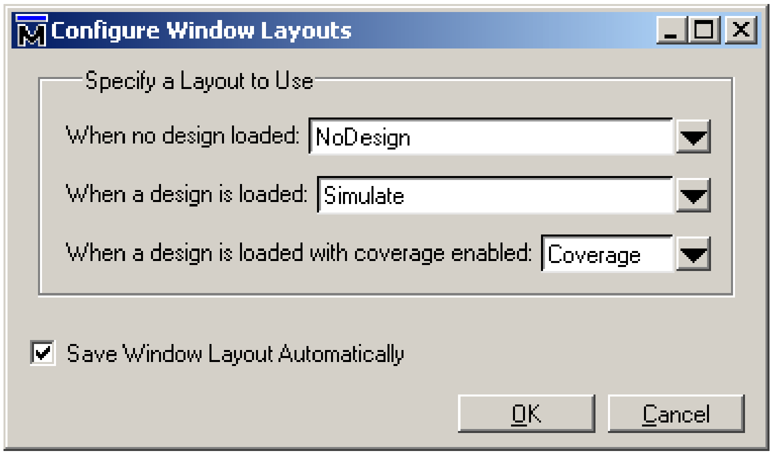
3. Specify a new name or use an existing name to overwrite that layout.
4. Click OK.

The layout is saved to the *.modelsim* file (or Registry on Windows).

Assigning Layouts to Modes

You can assign which layout appears in each mode (no design loaded, design loaded, design loaded with coverage). Follow these steps:

1. Create your custom layouts as described above.
2. Select **Window** > **Layouts** > **Configure**.



Specify a Layout to Use

When no design loaded: NoDesign

When a design is loaded: Simulate

When a design is loaded with coverage enabled: Coverage

☑ Save Window Layout Automatically

3. Select a layout for each mode.
4. Click OK.

The layout assignment is saved to the *.modelsim* file (Registry on Windows).

Automatic Saving of Layouts

By default any changes you make to a layout are saved automatically when you exit the tool or when you change modes. For example, if you load a design with code coverage, rearrange some windows, and then quit the simulation, the changes are saved to whatever layout was assigned to the "load with coverage" mode.

To disable automatic saving of layouts, select **Window** > **Layouts** > **Configure** and uncheck **Save Window Layout Automatically**.

Resetting Layouts to Their Defaults

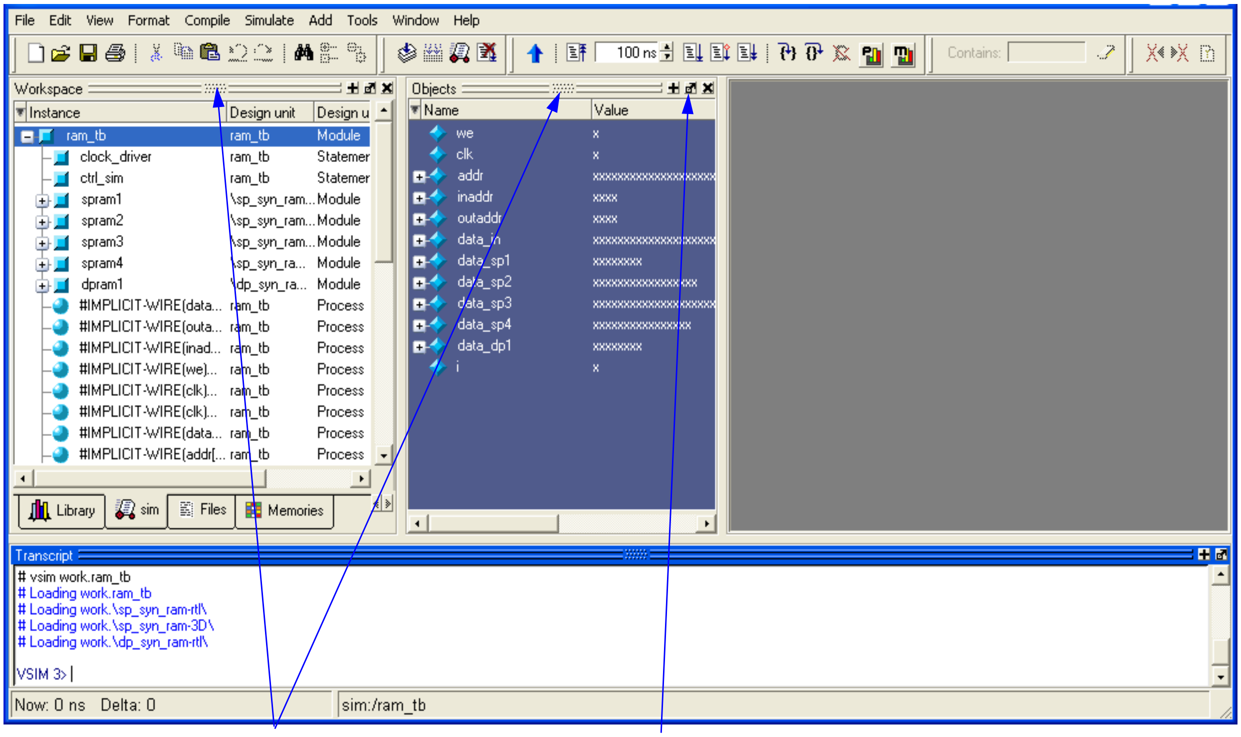
You can reset the layouts for the three modes to their original defaults. Select **Window** > **Layouts** > **Reset**. This command does not delete custom layouts.

Navigating the Graphic User Interface

This section discusses how to rearrange various elements of the GUI.

Moving, Docking, and Undocking Panes

Window panes (e.g., Transcript) can be positioned at various places within the parent window or they can be dragged out ("undocked") of the parent window altogether. When you see a double bar at the top edge of a pane, it means you can modify the pane position.



Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns Delta: 0 sim:/ram_tb

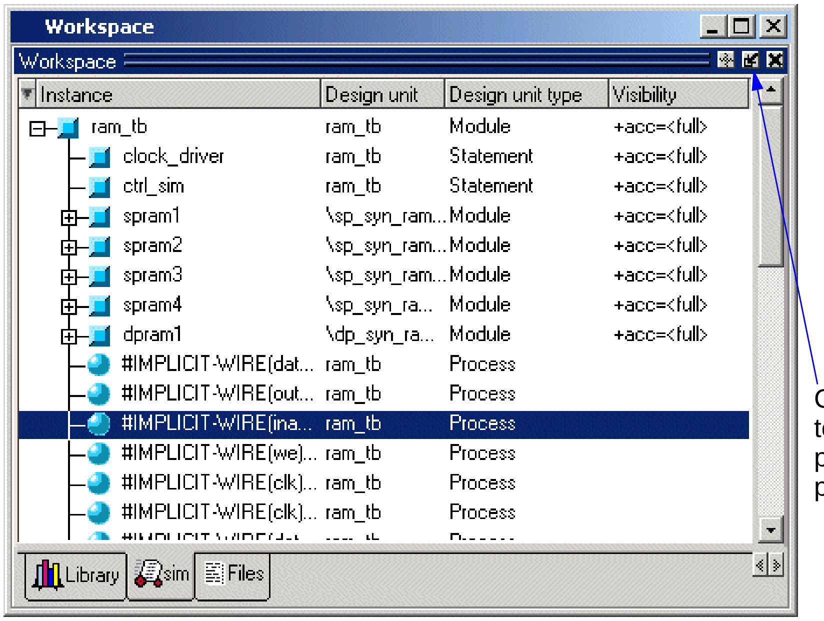
Click-and-drag in the middle of a double bar to move, undock, or dock a pane

Click this icon to undock a pane; click it again to redock

Click-and drag the pane handle in the middle of a double bar (your mouse pointer will change to a four-headed arrow when it is in the correct location) to reposition the pane inside the parent window. As you move the mouse to various parts of the main window, a gray outline will show you valid locations to drop the pane.

Or, drag the pane outside of the parent window, and when you let go of the mouse button, the pane becomes a free-floating window.

To redock a floating pane, click on the pane handle at the top of the window and drag it back into the parent window, or click the undock/dock icon as shown in the graphic below:



Instance	Design unit	Design unit type	Visibility
ram_tb	ram_tb	Module	+acc=<full>
clock_driver	ram_tb	Statement	+acc=<full>
ctrl_sim	ram_tb	Statement	+acc=<full>
spram1	\sp_syn_ram...	Module	+acc=<full>
spram2	\sp_syn_ram...	Module	+acc=<full>
spram3	\sp_syn_ram...	Module	+acc=<full>
spram4	\sp_syn_ra...	Module	+acc=<full>
dpram1	\dp_syn_ra...	Module	+acc=<full>
#IMPLICIT-WIRE(dat...	ram_tb	Process	
#IMPLICIT-WIRE(out...	ram_tb	Process	
#IMPLICIT-WIRE(ina...	ram_tb	Process	
#IMPLICIT-WIRE(we)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	

Click this icon to redock a pane in its parent window

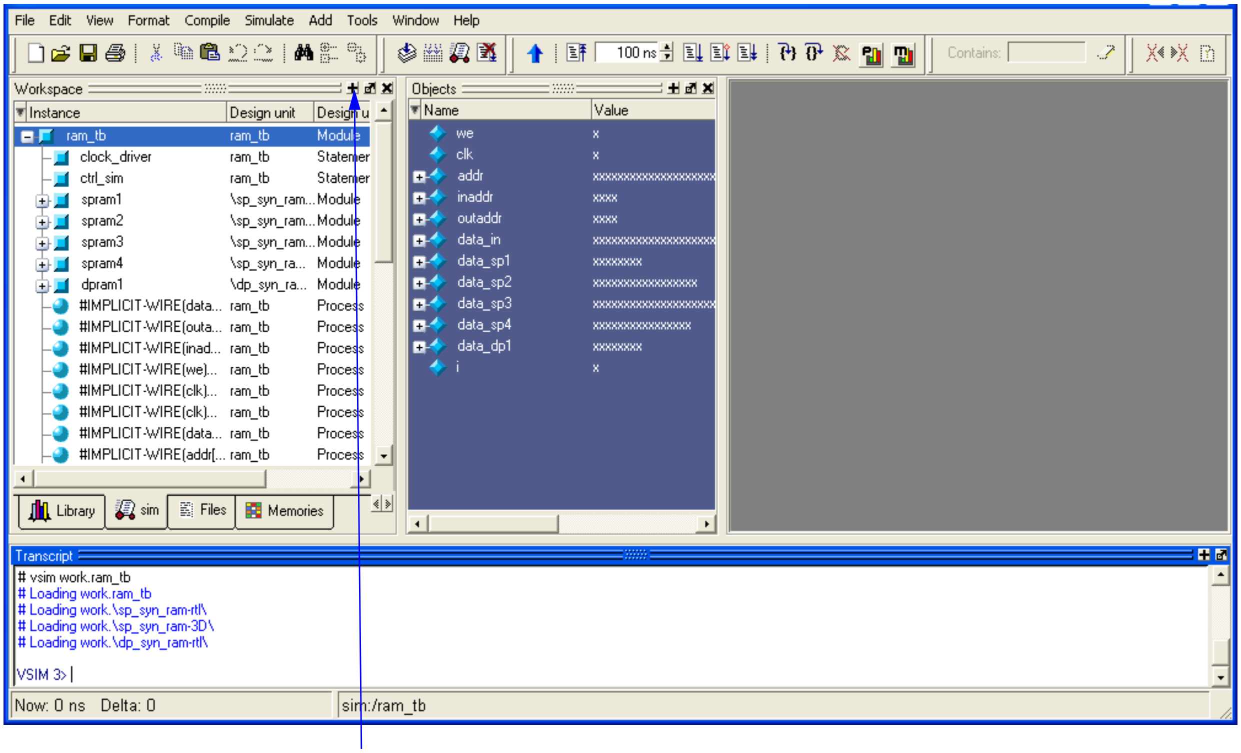
You can also undock a pane by clicking the undock/dock icon, as noted in the picture above.

Note

If you want to return to the original default layouts, select **Window** > **Layouts** > **Reset**.

Zooming Panes

You can expand panes to fill the entire Main window by clicking the zoom icon. For example, in the graphic below, clicking the zoom icon on the Workspace pane makes it fill the entire Main window, as shown on the following page.



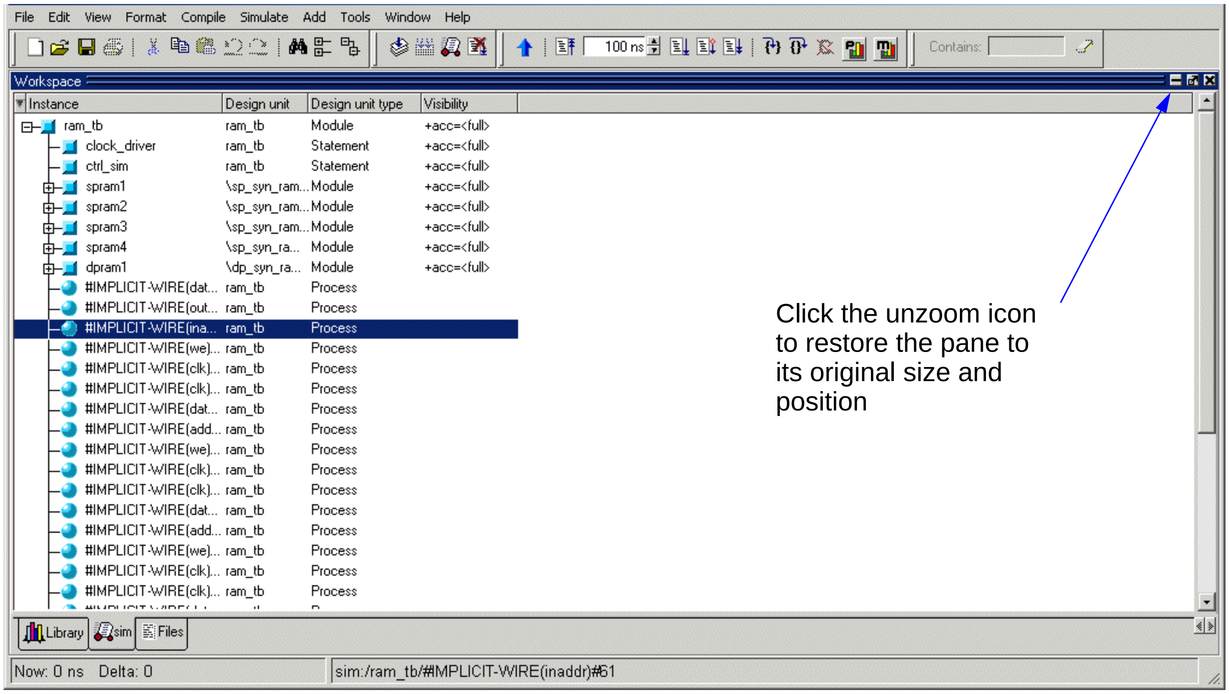
Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns   Delta: 0   sim:/ram_tb

Click the zoom icon to expand a pane to fill the entire window



Instance	Design unit	Design unit type	Visibility
ram_tb	ram_tb	Module	+acc=<full>
clock_driver	ram_tb	Statement	+acc=<full>
ctrl_sim	ram_tb	Statement	+acc=<full>
spram1	\sp_syn_ram...	Module	+acc=<full>
spram2	\sp_syn_ram...	Module	+acc=<full>
spram3	\sp_syn_ram...	Module	+acc=<full>
spram4	\sp_syn_ra...	Module	+acc=<full>
dpram1	\dp_syn_ra...	Module	+acc=<full>
#IMPLICIT-WIRE(dat...	ram_tb	Process	
#IMPLICIT-WIRE(out...	ram_tb	Process	
#IMPLICIT-WIRE(ina...	ram_tb	Process	
#IMPLICIT-WIRE(we)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(dat...	ram_tb	Process	
#IMPLICIT-WIRE(add...	ram_tb	Process	
#IMPLICIT-WIRE(we)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(dat...	ram_tb	Process	
#IMPLICIT-WIRE(add...	ram_tb	Process	
#IMPLICIT-WIRE(we)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	
#IMPLICIT-WIRE(clk)...	ram_tb	Process	

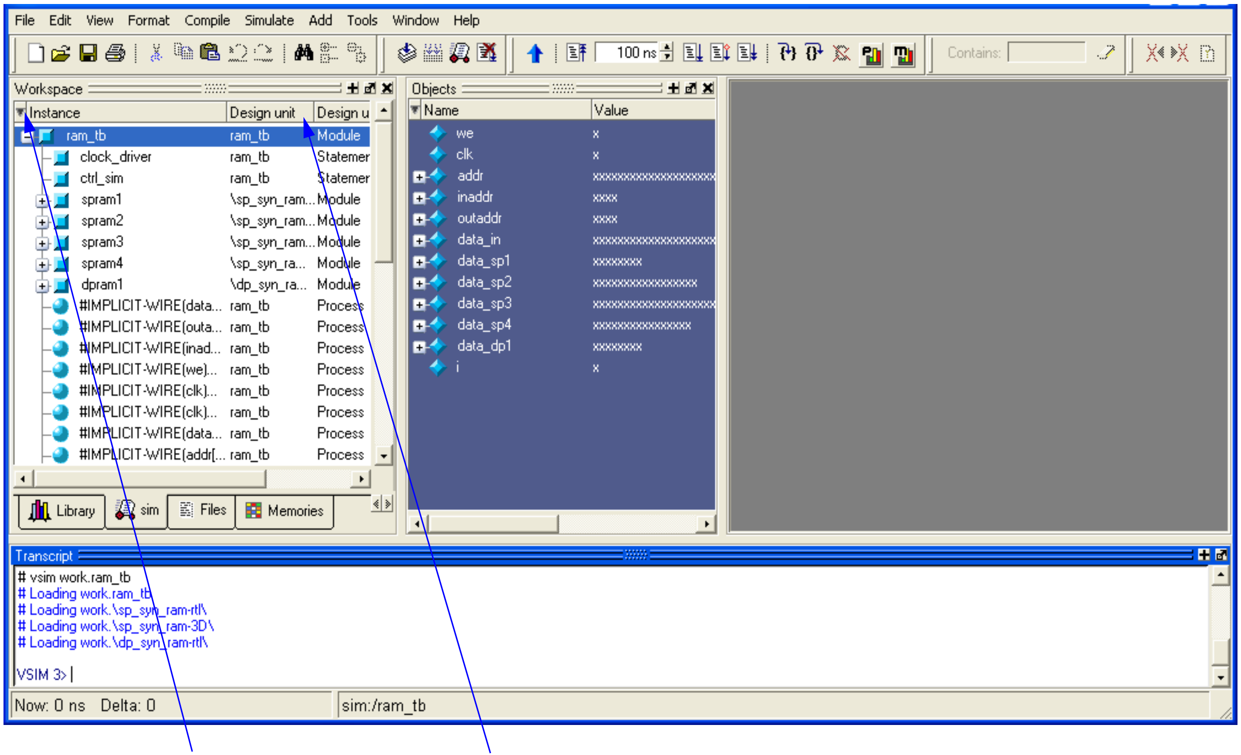
Now: 0 ns Delta: 0 | sim:/ram_tb/#IMPLICIT-WIRE(inaddr)#61

Click the unzoom icon to restore the pane to its original size and position

Columnar Information Display

Many panes (e.g., Objects, Workspace, etc.) display information in a columnar format. You can perform a number of operations on columnar formats:

- Click and drag on a column heading to rearrange columns
- Click and drag on a border between column names to increase/decrease column size
- Sort columns by clicking once on the column heading to sort in ascending order; clicking twice to sort in descending order; and clicking three times to sort in default order.
- Hide or show columns by either right-clicking a column heading and selecting an object from the context menu or by clicking the column-list drop down arrow and selecting an object.



Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns Delta: 0 | sim:/ram_tb

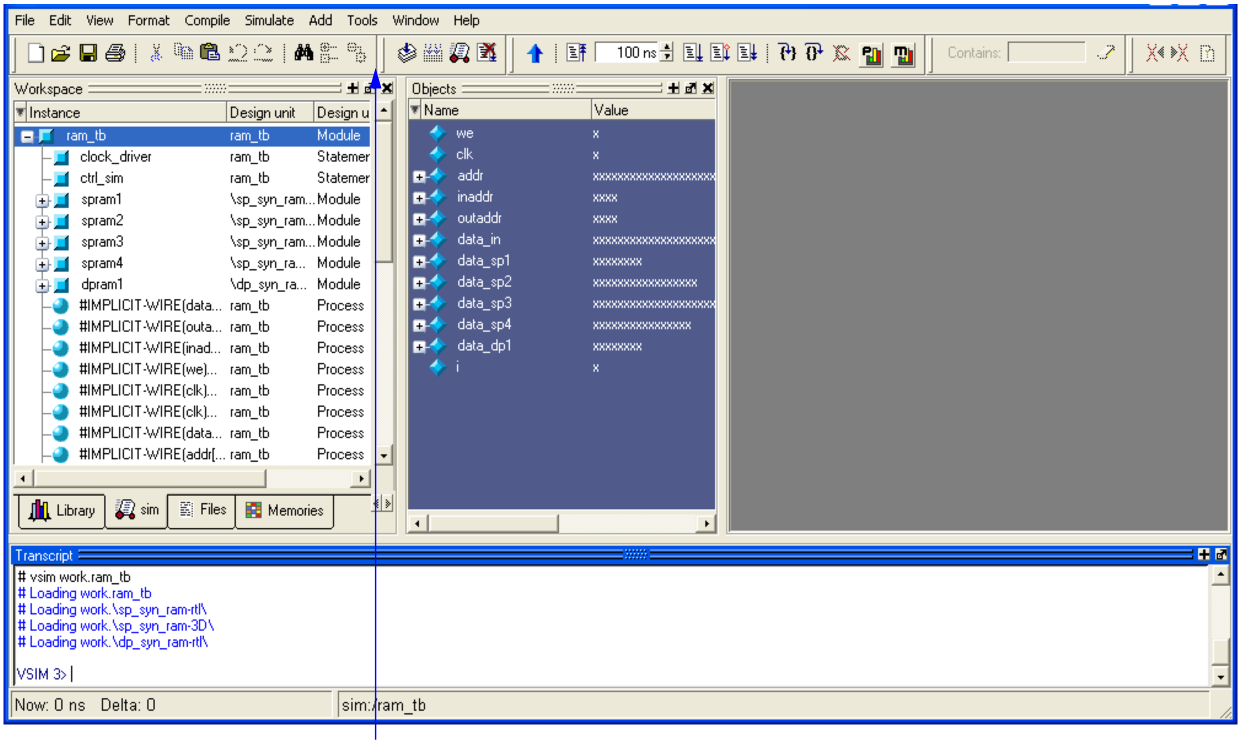
Click the down arrow to hide/show columns

Click on a column to sort the list

Quick Access Toolbars

Toolbar buttons provide access to commonly used commands and functions. Toolbars can be docked and undocked (moved to or from the main toolbar area) by clicking and dragging on the vertical bar at the left-edge of a toolbar.

You can also hide/show the various toolbars. To hide or show a toolbar, right-click on a blank spot of the main toolbar area and select a toolbar from the list.



Instance	Design unit	Design u
ram_tb	ram_tb	Module
clock_driver	ram_tb	Statemer
ctrl_sim	ram_tb	Statemer
spram1	\sp_syn_ram...	Module
spram2	\sp_syn_ram...	Module
spram3	\sp_syn_ram...	Module
spram4	\sp_syn_ra...	Module
dpram1	\dp_syn_ra...	Module
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(outa...	ram_tb	Process
#IMPLICIT-WIRE(inad...	ram_tb	Process
#IMPLICIT-WIRE(we)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(clk)...	ram_tb	Process
#IMPLICIT-WIRE(data...	ram_tb	Process
#IMPLICIT-WIRE(addr[...	ram_tb	Process

Name	Value
we	x
clk	x
addr	xxxxxxxxxxxxxxxxxxxx
inaddr	xxxx
outaddr	xxxx
data_in	xxxxxxxxxxxxxxxxxxxx
data_sp1	xxxxxxxx
data_sp2	xxxxxxxxxxxxxxxxx
data_sp3	xxxxxxxxxxxxxxxxxxxx
data_sp4	xxxxxxxxxxxxxxxx
data_dp1	xxxxxxxx
i	x

```
# vsim work.ram_tb
# Loading work.ram_tb
# Loading work.\sp_syn_ram-rtl\
# Loading work.\sp_syn_ram-3D\
# Loading work.\dp_syn_ram-rtl\

VSIM 3> |
```

Now: 0 ns   Delta: 0   sim:/ram_tb

Drag on the vertical
bar to dock/undock or
rearrange a toolbar

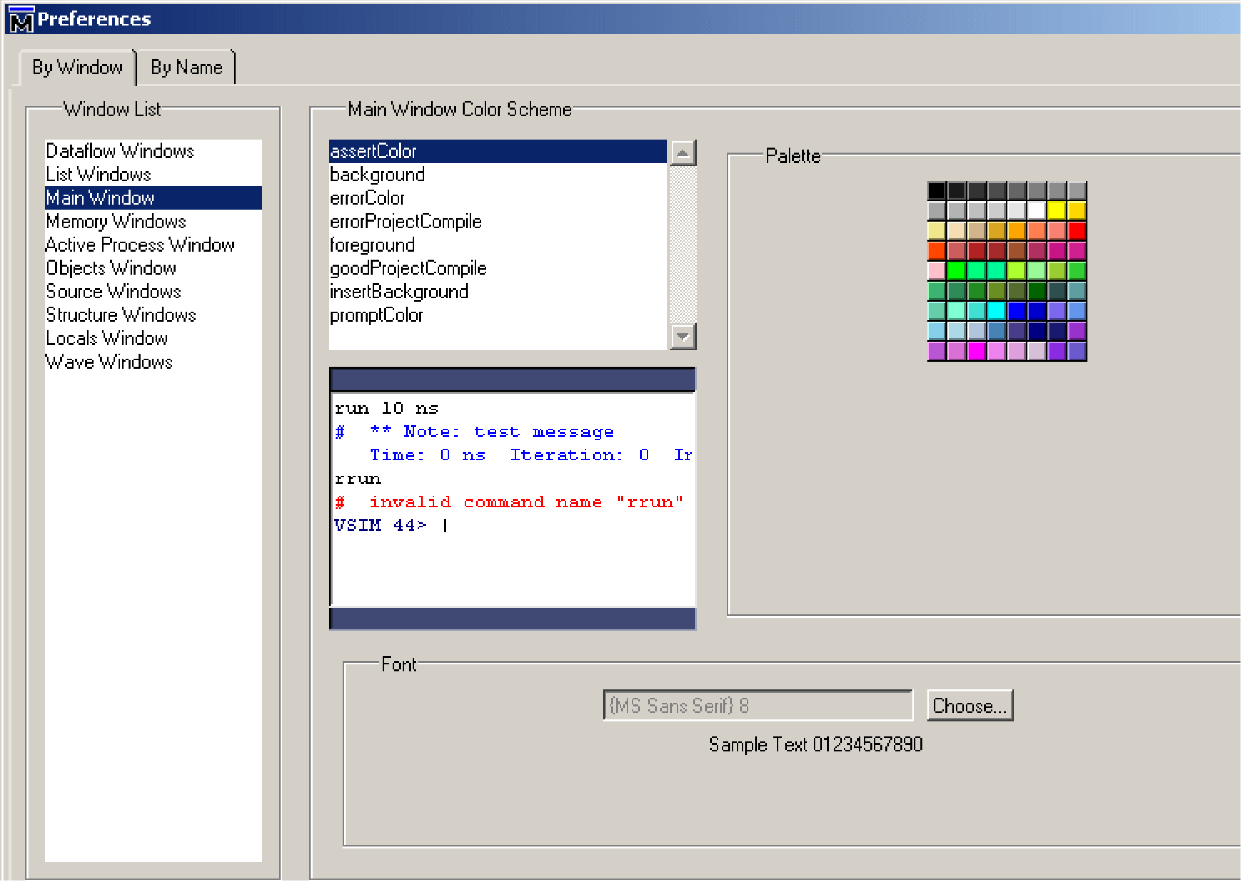
To reset toolbars to their original state, right-click on a blank spot of the main toolbar area and select **Reset**.

Simulator GUI Preferences

Simulator GUI preferences are stored by default either in the *.modelsim* file in your HOME directory on UNIX/Linux platforms or the Registry on Windows platforms.

Setting Preference Variables from the GUI

To edit a variable value from the GUI, select **Tools > Edit Preferences**.



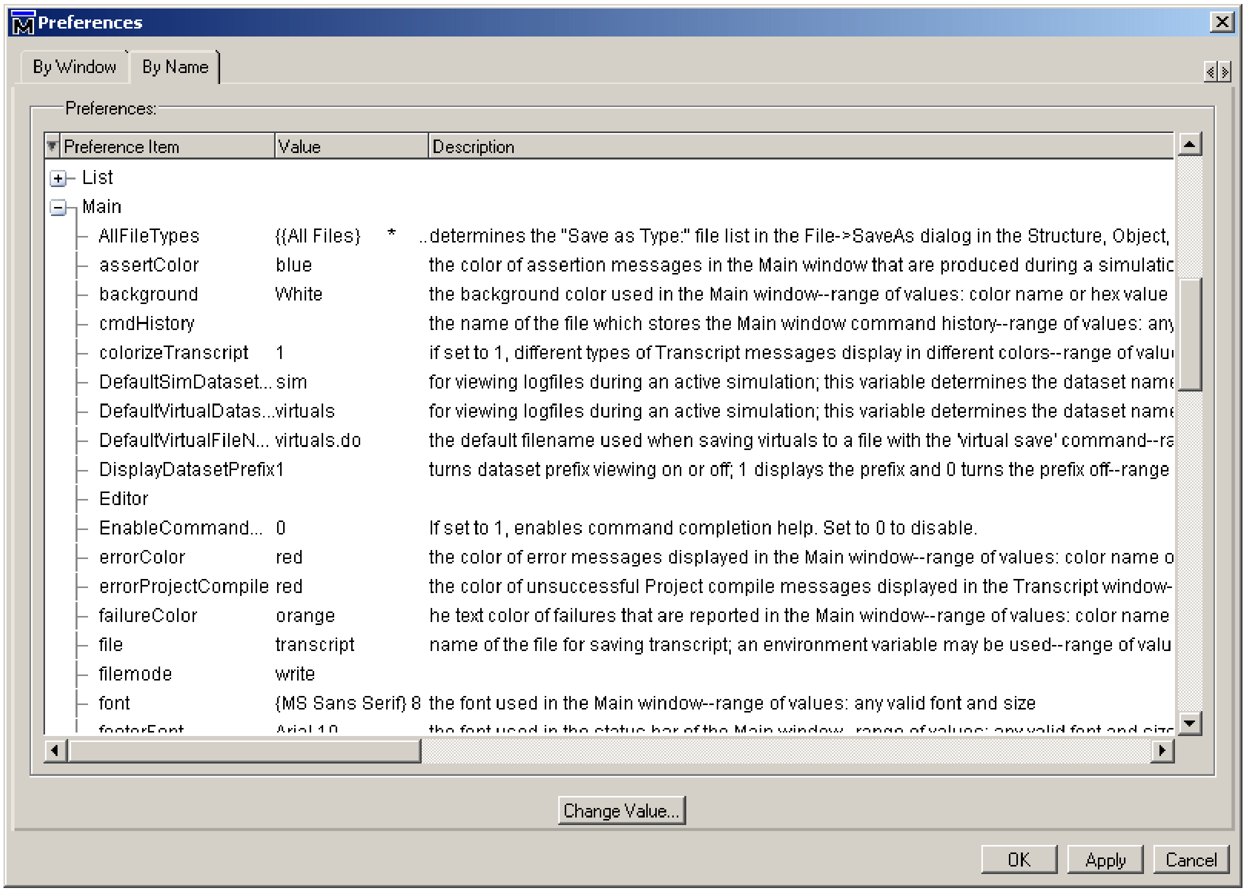
```
run 10 ns
#  ** Note: test message
   Time: 0 ns  Iteration: 0  Ir
rrun
#  invalid command name "rrun"
VSIM 44> |
```

{MS Sans Serif} 8

Sample Text 01234567890

The dialog organizes preferences by window and by name. The By Window tab primarily allows you to change colors and fonts for various GUI objects. For example, if you want to change the color of assertion messages in the Main window, you would select "Main window" in the first column, select "assertColor" in the second column, and click a color on the palette. Clicking OK or Apply changes the variable, and the change is saved when you exit ModelSim.

The By Name tab lists every Tcl variable in a tree structure. Expand the tree, highlight a variable, and click Change Value to edit the current value..



Preference Item	Value	Description
List		
Main		
AllFileTypes	{{All Files} *	..determines the "Save as Type:" file list in the File->SaveAs dialog in the Structure, Object,
assertColor	blue	the color of assertion messages in the Main window that are produced during a simulatic
background	White	the background color used in the Main window--range of values: color name or hex value
cmdHistory		the name of the file which stores the Main window command history--range of values: any
colorizeTranscript	1	if set to 1, different types of Transcript messages display in different colors--range of valu
DefaultSimDataset...	sim	for viewing logfiles during an active simulation; this variable determines the dataset name
DefaultVirtualDatas...	virtuals	for viewing logfiles during an active simulation; this variable determines the dataset name
DefaultVirtualFileN...	virtuals.do	the default filename used when saving virtuals to a file with the 'virtual save' command--ra
DisplayDatasetPrefix	1	turns dataset prefix viewing on or off; 1 displays the prefix and 0 turns the prefix off--range
Editor		
EnableCommand...	0	If set to 1, enables command completion help. Set to 0 to disable.
errorColor	red	the color of error messages displayed in the Main window--range of values: color name o
errorProjectCompile	red	the color of unsuccessful Project compile messages displayed in the Transcript window-
failureColor	orange	he text color of failures that are reported in the Main window--range of values: color name
file	transcript	name of the file for saving transcript; an environment variable may be used--range of valu
filemode	write	
font	{MS Sans Serif} 8	the font used in the Main window--range of values: any valid font and size
footerFont	Arial 10	the font used in the status bar of the Main window--range of values: any valid font and size

Setting Preference Variables from the Command Line

Use the Tcl set Command Syntax to customize preference variables from the Main window command line. For example:

set <variable name> <variable value>

Saving GUI Preferences

GUI preferences are saved automatically when you exit the tool.

If you prefer to store GUI preferences elsewhere, set the MODELSIM_PREFERENCES environment variable to designate where these preferences are stored. Setting this variable causes ModelSim to use a specified path and file instead of the default location. Here are some additional points to keep in mind about this variable setting:

- The file does not need to exist before setting the variable as ModelSim will initialize it.

- If the file is read-only, ModelSim will not update or otherwise modify the file.
- This variable may contain a relative pathname, in which case the file is relative to the working directory at the time the tool is started.

The modelsim.tcl File

Previous versions saved user GUI preferences into a *modelsim.tcl* file. Current versions will still read in a *modelsim.tcl* file if it exists. ModelSim searches for the file as follows:

- use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else
- use *./modelsim.tcl*; else
- use $(HOME)/modelsim.tcl if it exists

Note that in versions 6.1 and later, ModelSim will save to the *.modelsim* file any variables it reads in from a *modelsim.tcl* file (except for user_hook variables). The values from the *modelsim.tcl* file will override like variables in the *.modelsim* file.

User_hook Variables

User_hook variables allow you to add buttons and menus to the GUI. User_hook variables can only be stored in a *modelsim.tcl* file. They are not stored in *.modelsim*. If you need to use user_hook variables, you must create a *modelsim.tcl* file to store them.

Appendix G
System Initialization

ModelSim goes through numerous steps as it initializes the system during startup. It accesses various files and environment variables to determine library mappings, configure the GUI, check licensing, and so forth.

Files Accessed During Startup

The table below describes the files that are read during startup. They are listed in the order in which they are accessed.

Table G-1.

File	**Purpose**
modelsim.ini	contains initial tool settings; see Control Variables Located in INI Files for specific details on the *modelsim.ini* file
location map file	used by ModelSim tools to find source files based on easily reallocated "soft" paths; default file name is mgc_location_map
pref.tcl	contains defaults for fonts, colors, prompts, window positions, and other simulator window characteristics
.modelsim (UNIX) or Windows registry	contains last working directory, project file, printer defaults, and other user-customized GUI characteristics
modelsim.tcl	contains user-customized settings for fonts, colors, prompts, other GUI characteristics; maintained for backwards compatibility with older versions (see The modelsim.tcl File)
<project_name>.mpf	if available, loads last project file which is specified in the registry (Windows) or *$(HOME)/.modelsim* (UNIX); see What are Projects? for details on project settings

Environment Variables Accessed During Startup

The table below describes the environment variables that are read during startup. They are listed in the order in which they are accessed. For more information on environment variables, see Environment Variables.

Table G-2.

Environment variable	Purpose
MODEL_TECH	set by ModelSim to the directory in which the binary executables reside (e.g., *../modeltech/<platform>/*)
MODEL_TECH_OVERRIDE	provides an alternative directory for the binary executables; MODEL_TECH is set to this path
MODELSIM	identifies the pathname of the *modelsim.ini* file
MGC_WD	identifies the Mentor Graphics working directory
MGC_LOCATION_MAP	identifies the pathname of the location map file; set by ModelSim if not defined
MODEL_TECH_TCL	identifies the pathname of all Tcl libraries installed with ModelSim
HOME	identifies your login directory (UNIX only)
MGC_HOME	identifies the pathname of the MGC tool suite
TCL_LIBRARY	identifies the pathname of the Tcl library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
TK_LIBRARY	identifies the pathname of the Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITCL_LIBRARY	identifies the pathname of the [incr]Tcl library; set by ModelSim to the same path as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
ITK_LIBRARY	identifies the pathname of the [incr]Tk library; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
VSIM_LIBRARY	identifies the pathname of the Tcl files that are used by ModelSim; set by ModelSim to the same pathname as MODEL_TECH_TCL; must point to libraries supplied by Model Technology
MTI_COSIM_TRACE	creates an *mti_trace_cosim* file containing debugging information about FLI/PLI/VPI function calls; set to any value before invoking the simulator
MTI_LIB_DIR	identifies the path to all Tcl libraries installed with ModelSim

Table G-2.

Environment variable	Purpose
MTI_VCO_MODE	determines which version of ModelSim to use on platforms that support both 32- and 64-bit versions when ModelSim executables are invoked from the *modeltech/bin* directory by a Unix shell command (using full path specification or PATH search)
MODELSIM_TCL	identifies the pathname to a user preference file (e.g., *C:\modeltech\modelsim.tcl*); can be a list of file pathnames, separated by semicolons (Windows) or colons (UNIX); note that user preferences are now stored in the *.modelsim* file (Unix) or registry (Windows); ModelSim will still read this environment variable but it will then save all the settings to the *.modelsim* file when you exit the tool

Initialization Sequence

The following list describes in detail ModelSim's initialization sequence. The sequence includes a number of conditional structures, the results of which are determined by the existence of certain files and the current settings of environment variables.

In the steps below, names in uppercase denote environment variables (except MTI_LIB_DIR which is a Tcl variable). Instances of *$(NAME)* denote paths that are determined by an environment variable (except *$(MTI_LIB_DIR)* which is determined by a Tcl variable).

1. Determines the path to the executable directory (*../modeltech/<platform>*). Sets MODEL_TECH to this path, *unless* MODEL_TECH_OVERRIDE exists, in which case MODEL_TECH is set to the same value as MODEL_TECH_OVERRIDE.
2. Finds the *modelsim.ini* file by evaluating the following conditions:
   - use MODELSIM if it exists; else
   - use *$(MGC_WD)/modelsim.ini*; else
   - use *./modelsim.ini*; else
   - use *$(MODEL_TECH)/modelsim.ini*; else
   - use *$(MODEL_TECH)/../modelsim.ini*; else
   - use *$(MGC_HOME)/lib/modelsim.ini*; else
   - set path to *./modelsim.ini* even though the file doesn't exist
3. Finds the location map file by evaluating the following conditions:
   - use MGC_LOCATION_MAP if it exists (if this variable is set to "no_map", ModelSim skips initialization of the location map); else

- use *mgc_location_map* if it exists; else
- use *$(HOME)/mgc/mgc_location_map*; else
- use *$(HOME)/mgc_location_map*; else
- use *$(MGC_HOME)/etc/mgc_location_map*; else
- use *$(MGC_HOME)/shared/etc/mgc_location_map*; else
- use *$(MODEL_TECH)/mgc_location_map*; else
- use *$(MODEL_TECH)/../mgc_location_map*; else
- use no map

4. Reads various variables from the [vsim] section of the *modelsim.ini* file. See Simulator Control Variables for more details.
5. Parses any command line arguments that were included when you started ModelSim and reports any problems.
6. Defines the following environment variables:
   - use MODEL_TECH_TCL if it exists; else
   - set MODEL_TECH_TCL=*$(MODEL_TECH)/../tcl*
   - set TCL_LIBRARY=*$(MODEL_TECH_TCL)/tcl8.3*
   - set TK_LIBRARY=*$(MODEL_TECH_TCL)/tk8.3*
   - set ITCL_LIBRARY=*$(MODEL_TECH_TCL)/itcl3.0*
   - set ITK_LIBRARY=*$(MODEL_TECH_TCL)/itk3.0*
   - set VSIM_LIBRARY=*$(MODEL_TECH_TCL)/vsim*
7. Initializes the simulator's Tcl interpreter.
8. Checks for a valid license (a license is not checked out unless specified by a *modelsim.ini* setting or command line option).
9. The next four steps relate to initializing the graphical user interface.
10. Sets Tcl variable MTI_LIB_DIR=$(MODEL_TECH_TCL)
11. Loads *$(MTI_LIB_DIR)/vsim/pref.tcl*.
12. Loads gui preferences, project file, etc. from the registry (Windows) or *$(HOME)/.modelsim* (UNIX).
13. Searches for the *modelsim.tcl* file by evaluating the following conditions:
    - use MODELSIM_TCL environment variable if it exists (if MODELSIM_TCL is a list of files, each file is loaded in the order that it appears in the list); else

- use *./modelsim.tcl*; else
- use *$(HOME)/modelsim.tcl* if it exists

That completes the initialization sequence. Also note the following about the *modelsim.ini* file:

- When you change the working directory within ModelSim, the tool reads the [library], [vcom], and [vlog] sections of the local *modelsim.ini* file. When you make changes in the compiler or simulator options dialog or use the **vmap** command, the tool updates the appropriate sections of the file.
- The *pref.tcl* file references the default .ini file via the [GetPrivateProfileString] Tcl command. The .ini file that is read will be the default file defined at the time *pref.tcl* is loaded.

Appendix H
Logic Modeling Hardware Models

Logic Modeling hardware models can be used with ModelSim VHDL and Verilog. A hardware model allows simulation of a device using the actual silicon installed as a hardware model in one of Logic Modeling's hardware modeling systems. The hardware modeling system is a network resource with a procedural interface that is accessed by the simulator. This appendix describes how to use Logic Modeling hardware models with ModelSim.

Note

Please refer to Logic Modeling documentation from Synopsys for details on using the hardware modeler. This appendix only describes the specifics of using hardware models with ModelSim.

VHDL Hardware Model Interface

ModelSim VHDL interfaces to a hardware model through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific hardware model with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the hardware modeler software and establishes communication with the specific hardware model.

The ModelSim software locates the hardware modeler interface software based on entries in the *modelsim.ini* initialization file. The simulator and the **hm_entity** tool (for creating foreign architectures) both depend on these entries being set correctly. These entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
libhm = $MODEL_TECH/libhm.sl
; ModelSim's interface to Logic Modeling's hardware modeler SFI software
(Windows NT)
; libhm = $MODEL_TECH/libhm.dll
;  Logic Modeling's hardware modeler SFI software (HP 9000 Series 700)
; libsfi = <sfi_dir>/lib/hp700/libsfi.sl
;  Logic Modeling's hardware modeler SFI software (IBM RISC System/6000)
; libsfi = <sfi_dir>/lib/rs6000/libsfi.a
;  Logic Modeling's hardware modeler SFI software (Sun4 Solaris)
; libsfi = <sfi_dir>/lib/sun4.solaris/libsfi.so
;  Logic Modeling's hardware modeler SFI software (Window NT)
; libsfi = <sfi_dir>/lib/pcnt/lm_sfi.dll
;  Logic Modeling's hardware modeler SFI software (Linux)
; libsfi = <sfi_dir>/lib/linux/libsfi.so
```

The **libhm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the hardware modeler software. The **libsfi** entry points to the Logic Modeling dynamic link library software that accesses the hardware modeler. The simulator automatically loads both the **libhm** and **libsfi** libraries when it elaborates a hardware model foreign architecture.

By default, the **libhm** entry points to the *libhm.sl* supplied in the ModelSim installation directory indicated by the MODEL_TECH environment variable. ModelSim automatically sets the MODEL_TECH environment variable to the appropriate directory containing the executables and binaries for the current operating system. If you are running the Windows operating system, then you must comment out the default **libhm** entry (precede the line with the ";" character) and uncomment the **libhm** entry for the Windows operating system.

Uncomment the appropriate **libsfi** entry for your operating system, and replace **<sfi_dir>** with the path to the hardware modeler software installation directory. In addition, you must set the **LM_LIB** and **LM_DIR** environment variables as described in Logic Modeling documentation from Synopsys.

Creating Foreign Architectures with hm_entity

The ModelSim **hm_entity** tool automatically creates entities and foreign architectures for hardware models. Its usage is as follows:

Syntax

```
hm_entity
```

[-xe] [-xa] [-c] [-93] <shell software filename>

Arguments

`-xe`

Do not generate entity declarations.

`-xa`

Do not generate architecture bodies.

`-c`

Generate component declarations.

`-93`

Use extended identifiers where needed.

`<shell software filename>`

Hardware model shell software filename (see Logic Modeling documentation from Synopsys for details on shell software files)

By default, the **hm_entity** tool writes an entity and foreign architecture to stdout for the hardware model. Optionally, you can include the component declaration (-**c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

Once you have created the entity and foreign architecture, you must compile it into a library. For example, the following commands compile the entity and foreign architecture for a hardware model named **LMTEST**:

```
% hm_entity LMTEST.MDL > lmtest.vhd
% vlib lmc
% vcom -work lmc lmtest.vhd
```

To instantiate the hardware model in your VHDL design, you will also need to generate a component declaration. If you have multiple hardware models, you may want to add all of their component declarations to a package so that you can easily reference them in your design. The following command writes the component declaration to stdout for the **LMTEST** hardware model.

```
% hm_entity -c -xe -xa LMTEST.MDL
```

Paste the resulting component declaration into the appropriate place in your design or into a package.

The following is an example of the entity and foreign architecture created by **hm_entity** for the CY7C285 hardware model:

```
library ieee;
use ieee.std_logic_1164.all;

entity cy7c285 is
   generic ( DelayRange : STRING := "Max" );
   port ( A0 : in std_logic;
      A1 : in std_logic;
      A2 : in std_logic;
      A3 : in std_logic;
      A4 : in std_logic;
      A5 : in std_logic;
      A6 : in std_logic;
      A7 : in std_logic;
      A8 : in std_logic;
      A9 : in std_logic;
      A10 : in std_logic;
      A11 : in std_logic;
      A12 : in std_logic;
      A13 : in std_logic;
      A14 : in std_logic;
      A15 : in std_logic;
      CS : in std_logic;
      O0 : out std_logic;
      O1 : out std_logic;
      O2 : out std_logic;
      O3 : out std_logic;
      O4 : out std_logic;
      O5 : out std_logic;
      O6 : out std_logic;
      O7 : out std_logic;
      W : inout std_logic );
end;
```

```
architecture Hardware of cy7c285 is
   attribute FOREIGN : STRING;
   attribute FOREIGN of Hardware : architecture is
      "hm_init $MODEL_TECH/libhm.sl ; CY7C285.MDL";
begin
end Hardware;
```

Hardware Model Entity Details

- The entity name is the hardware model name (you can manually change this name if you like).
- The port names are the same as the hardware model port names *(these names must not be changed*). If the hardware model port name is not a valid VHDL identifier, then **hm_entity** issues an error message. If **hm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. Another option is to create a pin-name mapping file. Consult the Logic Modeling documentation from Synopsys for details.
- The port types are **std_logic**. This data type supports the full range of hardware model logic states.
- The *DelayRange* generic selects minimum, typical, or maximum delay values. Valid values are "min", "typ", or "max" (the strings are not case-sensitive). The default is "max".

Hardware Model Architecture Details

- The first part of the foreign attribute string (hm_init) is the same for all hardware models.
- The second part (*$MODEL_TECH/libhm.sl*) is taken from the **libhm** entry in the initialization file, *modelsim.ini*.
- The third part (CY7C285.MDL) is the shell software filename. This name correlates the architecture with the hardware model at elaboration.

Hardware Model Vector Ports

The entities generated by **hm_entity** only contain single-bit ports, never vectored ports. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 hardware model:

```
component cy7c285
   generic ( DelayRange : STRING := "Max");
   port ( A : in std_logic_vector (15 downto 0);
      CS : in std_logic;
      O : out std_logic_vector (7 downto 0);
      WAIT_PORT : inout std_logic );
end component;

for all: cy7c285
   use entity work.cy7c285
   port map (A0 => A(0),
      A1 => A(1),
      A2 => A(2),
      A3 => A(3),
      A4 => A(4),
      A5 => A(5),
      A6 => A(6),
      A7 => A(7),
      A8 => A(8),
      A9 => A(9),
      A10 => A(10),
      A11 => A(11),
      A12 => A(12),
      A13 => A(13),
      A14 => A(14),
      A15 => A(15),
      CS => CS,
      O0 => O(0),
      O1 => O(1),
      O2 => O(2),
      O3 => O(3),
      O4 => O(4),
      O5 => O(5),
      O6 => O(6),
      O7 => O(7),
      WAIT_PORT => W );
```

Hardware Model Commands

The following simulator commands are available for hardware models. Refer to the Logic Modeling documentation from Synopsys for details on these operations.

```
lm_vectors on|off <instance_name> [<filename>]
```

Enable/disable test vector logging for the specified hardware model.

```
lm_measure_timing on|off <instance_name> [<filename>]
```

Enable/disable timing measurement for the specified hardware model.

```
lm_timing_checks on|off <instance_name>
```

Enable/disable timing checks for the specified hardware model.

```
lm_loop_patterns on|off <instance_name>
```

Enable/disable pattern looping for the specified hardware model.

```
lm_unknowns on|off <instance_name>
```

Enable/disable unknown propagation for the specified hardware model.

Appendix I
Logic Modeling SmartModels

The Logic Modeling SWIFT-based SmartModel library can be used with ModelSim VHDL and Verilog. The SmartModel library is a collection of behavioral models supplied in binary form with a procedural interface that is accessed by the simulator. This appendix describes how to use the SmartModel library with ModelSim.

The SmartModel library must be obtained from Logic Modeling along with the documentation that describes how to use it. This appendix only describes the specifics of using the library with ModelSim.

A 32-bit SmartModel will not run with a 64-bit version of SE. When trying to load the operating system specific 32-bit library into the 64-bit executable, the pointer sizes will be incorrect.

VHDL SmartModel Interface

ModelSim VHDL interfaces to a SmartModel through a foreign architecture. The foreign architecture contains a foreign attribute string that associates a specific SmartModel with the architecture. On elaboration of the foreign architecture, the simulator automatically loads the SmartModel library software and establishes communication with the specific SmartModel.

Enabling the Interface

To enable the SmartModel interface you must do the following:

- Set the **LMC_HOME** environment variable to the root of the SmartModel library installation directory. Consult Logic Modeling's documentation for details.
- Uncomment the appropriate **libswift** entry in the *modelsim.ini* file for your operating system.
- If you are running the Windows operating system, you must also comment out the default **libsm** entry (precede the line with the ";" character) and uncomment the **libsm** entry for the Windows operating system.

The **libswift** and **libsm** entries are found under the [lmc] section of the default *modelsim.ini* file located in the ModelSim installation directory. The default settings are as follows:

```
[lmc]
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
libsm = $MODEL_TECH/libsm.sl
; ModelSim's interface to Logic Modeling's SmartModel SWIFT software
(Windows NT)
; libsm = $MODEL_TECH/libsm.dll
;  Logic Modeling's SmartModel SWIFT software (HP 9000 Series 700)
; libswift = $LMC_HOME/lib/hp700.lib/libswift.sl
;  Logic Modeling's SmartModel SWIFT software (IBM RISC System/6000)
; libswift = $LMC_HOME/lib/ibmrs.lib/swift.o
;  Logic Modeling's SmartModel SWIFT software (Sun4 Solaris)
; libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
;  Logic Modeling's SmartModel SWIFT software (Windows NT)
; libswift = $LMC_HOME/lib/pcnt.lib/libswift.dll
;  Logic Modeling's SmartModel SWIFT software (Linux)
; libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
```

The **libsm** entry points to the ModelSim dynamic link library that interfaces the foreign architecture to the SmartModel software. The **libswift** entry points to the Logic Modeling dynamic link library software that accesses the SmartModels. The simulator automatically loads both the **libsm** and **libswift** libraries when it elaborates a SmartModel foreign architecture.

By default, the **libsm** entry points to the *libsm.sl* supplied in the ModelSim installation directory indicated by the **MODEL_TECH** environment variable. ModelSim automatically sets the **MODEL_TECH** environment variable to the appropriate directory containing the executables and binaries for the current operating system.

Creating Foreign Architectures with sm_entity

The ModelSim **sm_entity** tool automatically creates entities and foreign architectures for SmartModels. Its usage is as follows:

Syntax

`sm_entity`

[-] [-xe] [-xa] [-c] [-all] [-v] [-93] [<SmartModelName>...]

Arguments

`-`

Read SmartModel names from standard input.

`-xe`

Do not generate entity declarations.

`-xa`

Do not generate architecture bodies.

`-c`

Generate component declarations.

`-all`

Select all models installed in the SmartModel library.

`-v`
Display progress messages.

`-93`
Use extended identifiers where needed.

`<SmartModelName>`
Name of a SmartModel (see the SmartModel library documentation for details on SmartModel names).

By default, the **sm_entity** tool writes an entity and foreign architecture to stdout for each SmartModel name listed on the command line. Optionally, you can include the component declaration (**-c**), exclude the entity (**-xe**), and exclude the architecture (**-xa**).

The simplest way to prepare SmartModels for use with ModelSim VHDL is to generate the entities and foreign architectures for all installed SmartModels, and compile them into a library named **lmc**. This is easily accomplished with the following commands:

```
% sm_entity -all > sml.vhd
% vlib lmc
% vcom -work lmc sml.vhd
```

To instantiate the SmartModels in your VHDL design, you also need to generate component declarations for the SmartModels. Add these component declarations to a package named **sml** (for example), and compile the package into the **lmc** library:

```
% sm_entity -all -c -xe -xa > smlcomp.vhd
```

Edit the resulting *smlcomp.vhd* file to turn it into a package of SmartModel component declarations as follows:

```
library ieee;
use ieee.std_logic_1164.all;
package sml is
   <component declarations go here>
end sml;
```

Compile the package into the **lmc** library:

```
% vcom -work lmc smlcomp.vhd
```

The SmartModels can now be referenced in your design by adding the following **library** and **use** clauses to your code:

```
library lmc;
use lmc.sml.all;
```

The following is an example of an entity and foreign architecture created by **sm_entity** for the cy7c285 SmartModel.

```
library ieee;
use ieee.std_logic_1164.all;

entity cy7c285 is
   generic (TimingVersion : STRING := "CY7C285-65";
      DelayRange : STRING := "Max";
      MemoryFile : STRING := "memory" );
   port ( A0 : in std_logic;
      A1 : in std_logic;
      A2 : in std_logic;
      A3 : in std_logic;
      A4 : in std_logic;
      A5 : in std_logic;
      A6 : in std_logic;
      A7 : in std_logic;
      A8 : in std_logic;
      A9 : in std_logic;
      A10 : in std_logic;
      A11 : in std_logic;
      A12 : in std_logic;
      A13 : in std_logic;
      A14 : in std_logic;
      A15 : in std_logic;
      CS : in std_logic;
      O0 : out std_logic;
      O1 : out std_logic;
      O2 : out std_logic;
      O3 : out std_logic;
      O4 : out std_logic;
      O5 : out std_logic;
      O6 : out std_logic;
      O7 : out std_logic;
      WAIT_PORT : inout std_logic );
end;
architecture SmartModel of cy7c285 is
   attribute FOREIGN : STRING;
   attribute FOREIGN of SmartModel : architecture is
      "sm_init $MODEL_TECH/libsm.sl ; cy7c285";
begin
end SmartModel;
```

SmartModel Entity Details

- The entity name is the SmartModel name (you can manually change this name if you like).
- The port names are the same as the SmartModel port names *(these names must not be changed*). If the SmartModel port name is not a valid VHDL identifier, then **sm_entity** automatically converts it to a valid name. If **sm_entity** is invoked with the **-93** option, then the identifier is converted to an extended identifier, and the resulting entity must also be compiled with the **-93** option. If the **-93** option had been specified in the example above, then *WAIT* would have been converted to *\WAIT*. Note that in this example the port *WAIT* was converted to *WAIT_PORT* because **wait** is a VHDL reserved word.

- The port types are **std_logic**. This data type supports the full range of SmartModel logic states.
- The *DelayRange*, *TimingVersion*, and *MemoryFil*e generics represent the SmartModel attributes of the same name. Consult your SmartModel library documentation for a description of these attributes (and others). **Sm_entity** creates a generic for each attribute of the particular SmartModel. The default generic value is the default attribute value that the SmartModel has supplied to **sm_entity**.

SmartModel Architecture Details

- The first part of the foreign attribute string (sm_init) is the same for all SmartModels.
- The second part (*$MODEL_TECH/libsm.sl*) is taken from the **libsm** entry in the initialization file, *modelsim.ini*.
- The third part (cy7c285) is the SmartModel name. This name correlates the architecture with the SmartModel at elaboration.

SmartModel Vector Ports

The entities generated by **sm_entity** only contain single-bit ports, never vectored ports. This is necessary because ModelSim correlates entity ports with the SmartModel SWIFT interface by name. However, for ease of use in component instantiations, you may want to create a custom component declaration and component specification that groups ports into vectors. You can also rename and reorder the ports in the component declaration. You can also reorder the ports in the entity declaration, but you can't rename them!

The following is an example component declaration and specification that groups the address and data ports of the CY7C285 SmartModel:

```
component cy7c285
   generic ( TimingVersion : STRING := "CY7C285-65";
      DelayRange : STRING := "Max";
      MemoryFile : STRING := "memory" );
   port ( A : in std_logic_vector (15 downto 0);
      CS : in std_logic;
      O : out std_logic_vector (7 downto 0);
      WAIT_PORT : inout std_logic );
end component;
```

```
for all: cy7c285
   use entity work.cy7c285
   port map (A0 => A(0),
      A1 => A(1),
      A2 => A(2),
      A3 => A(3),
      A4 => A(4),
      A5 => A(5),
      A6 => A(6),
      A7 => A(7),
      A8 => A(8),
      A9 => A(9),
      A10 => A(10),
      A11 => A(11),
      A12 => A(12),
      A13 => A(13),
      A14 => A(14),
      A15 => A(15),
      CS => CS,
      O0 => O(0),
      O1 => O(1),
      O2 => O(2),
      O3 => O(3),
      O4 => O(4),
      O5 => O(5),
      O6 => O(6),
      O7 => O(7),
      WAIT_PORT => WAIT_PORT );
```

Command Channel

The command channel is a SmartModel feature that lets you invoke SmartModel specific commands. These commands are documented in the SmartModel library documentation from Synopsys. ModelSim provides access to the Command Channel from the command line. The form of a SmartModel command is:

lmc <instance_name>|-all "<SmartModel command>"

The **instance_name** argument is either a full hierarchical name or a relative name of a SmartModel instance. A relative name is relative to the current environment setting (see environment command). For example, to turn timing checks off for SmartModel */top/u1*:

lmc /top/u1 "SetConstraints Off"

Use **-all** to apply the command to all SmartModel instances. For example, to turn timing checks off for all SmartModel instances:

lmc -all "SetConstraints Off"

There are also some SmartModel commands that apply globally to the current simulation session rather than to models. The form of a SmartModel session command is:

lmcsession "<SmartModel session command>"

SmartModel Windows

Some models in the SmartModel library provide access to internal registers with a feature called SmartModel Windows. Refer to Logic Modeling's SmartModel library documentation (available on Synopsys' web site) for details on this feature. The simulator interface to this feature is described below.

Window names that are not valid VHDL or Verilog identifiers are converted to VHDL extended identifiers. For example, with a window named z1I10.GSR.OR, ModelSim will treat the name as \z1I10.GSR.OR\ (for all commands including lmcwin, add wave, and examine). You must then use that name in all commands. For example,

```
add wave /top/swift_model/\z1I10.GSR.OR\
```

Extended identifiers are case sensitive.

ReportStatus

The **ReportStatus** command displays model information, including the names of window registers. For example,

```
lmc /top/u1 ReportStatus
```

SmartModel Windows description:

```
WA "Read-Only (Read Only)"
WB "1-bit"
WC "64-bit"
```

This model contains window registers named *wa*, *wb*, and *wc*. These names can be used in subsequent window (**lmcwin**) commands.

SmartModel lmcwin Commands

The following window commands are supported after you have loaded a SmartModel:

- **lmcwin read** <window_instance> [-<radix>]
- **lmcwin write** <window_instance> <value>
- **lmcwin enable** <window_instance>
- **lmcwin disable** <window_instance>
- **lmcwin release** <window_instance>

Each command requires a window instance argument that identifies a specific model instance and window name. For example, */top/u1/wa* refers to window *wa* in model instance */top/u1*.

- lmcwin read

The **lmcwin read** command displays the current value of a window. The optional radix argument is **-binary**, **-decimal**, or **-hexadecimal** (these names can be abbreviated). The default is to display the value using the **std_logic** characters. For example, the following command displays the 64-bit window *wc* in hexadecimal:

```
lmcwin read /top/u1/wc -h
```

- lmcwin write

  The **lmcwin write** command writes a value into a window. The format of the value argument is the same as used in other simulator commands that take value arguments. For example, to write 1 to window *wb,* and all 1's to window *wc*:

  ```
  lmcwin write /top/u1/wb 1
  lmcwin write /top/u1/wc X"FFFFFFFFFFFFFFFF"
  ```

- lmcwin enable

  The **lmcwin enable** command enables continuous monitoring of a window. The specified window is added to the model instance as a signal (with the same name as the window) of type **std_logic** or **std_logic_vector**. This signal's values can then be referenced in simulator commands that read signal values, such as the add list command shown below. The window signal is continuously updated to reflect the value in the model. For example, to list window *wa*:

  ```
  lmcwin enable /top/u1/wa
  add list /top/u1/wa
  ```

- lmcwin disable

  The **lmcwin disable** command disables continuous monitoring of a window. The window signal is not deleted, but it no longer is updated when the model's window register changes value. For example, to disable continuous monitoring of window *wa*:

  ```
  lmcwin disable /top/u1/wa
  ```

- lmcwin release

  Some windows are actually nets, and the **lmcwin write** command behaves more like a continuous force on the net. The **lmcwin release** command disables the effect of a previous **lmcwin write** command on a window net.

Memory Arrays

A memory model usually makes the entire register array available as a window. In this case, the window commands operate only on a single element at a time. The element is selected as an array reference in the window instance specification. For example, to read element 5 from the window memory *mem*:

```
lmcwin read /top/u2/mem(5)
```

Omitting the element specification defaults to element 0. Also, continuous monitoring is limited to a single array element. The associated window signal is updated with the most recently enabled element for continuous monitoring.

Verilog SmartModel Interface

The SWIFT SmartModel library, beginning with release r40b, provides an optional library of Verilog modules and a PLI application that communicates between a simulator's PLI and the SWIFT simulator interface. The Logic Modeling documentation refers to this as the Logic Models to Verilog (LMTV) interface. To install this option, you must select the simulator type "Verilog" when you run Logic Modeling’s SmartInstall program.

Linking the LMTV Interface to the Simulator

Synopsys provides a dynamically loadable library that links ModelSim to the LMTV interface. See chapter 5, "Using MTI Verilog with Synopsys Models," in the "Simulator Configuration Guide for Synopsys Models" (available on Synopsys’ web site) for directions on how to link to this library.

Index

Third-Party Information

This section provides information on third-party software that may be included in the ModelSim SE product, including any additional license terms.

- This product may include Valgrind third-party software.

  ©Julian Seward. All rights reserved.

  THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- This product may use MinGW GCC third-party software.

  ©Red Hat, Inc. All rights reserved.

  ©Pipeline Associates, Inc. All rights reserved.

  ©Matthew Self. All rights reserved.

  ©National Research Council of Canada. All rights reserved.

  ©The Regents of the University of California.

  THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

  ©Free Software Foundation, Inc. All rights reserved.

  Refer to the license file in your install directory:

  <install_directory>/docs/legal/mingw_gcc.pdf

- This software application may include GNU GCC third-party software.

  © AT&T. All rights reserved.

  Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

  THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

  Refer to the license file in your install directory:

    <install_directory>/docs/legal/gnu_gcc.pdf

- This software application may include GNU GCC third-party software.

  © Doug Bell. All Rights Reserved.

  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

  Refer to the license file in your install directory:

      <install_directory>/docs/legal/gnu_gcc.pdf

- This software application may include GNU third-party software distributed by The Free Software Foundation.

  © Free Software Foundation.

  To view a copy of the GNU GPL, LGPL, Library, and Documentation licenses, refer to:

  http://www.fsf.org/licensing/licenses.

  Refer to the license file in your install directory:

      <install_directory>/docs/legal/gnu_gcc.pdf

- This software application may include GNU GCC third-party software.

  ©The Regents of the University of California. All rights reserved.

  THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

  Refer to the license file in your install directory:

      <install_directory>/docs/legal/gnu_gcc.pdf

- This product may include freeWrap open source software

  © Dennis R. LaBelle All Rights Reserved.

  Disclaimer of warranty: Licensor provides the software on an ``as is'' basis. Licensor does not warrant, guarantee, or make any representations regarding the use or results of the software with respect to it correctness, accuracy, reliability or performance. The entire risk of the use and performance of the software is assumed by licensee. ALL WARANTIES INCLUDING, WITHOUT LIMITATION, ANY WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR MERCHANTABILITY ARE HEREBY EXCLUDED.

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/terms_conditions/enduser.cfm

IMPORTANT INFORMATION

USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software between you, the end user, as an authorized representative of the company acquiring the license, and Mentor Graphics Corporation and Mentor Graphics (Ireland) Limited acting directly or through their subsidiaries (collectively "Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by you and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **GRANT OF LICENSE.** The software programs, including any updates, modifications, revisions, copies, documentation and design data ("Software"), are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; (c) for the license term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions.

2. **EMBEDDED SOFTWARE.** If you purchased a license to use embedded software development ("ESD") Software, if applicable, Mentor Graphics grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into your products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

3. **BETA CODE.** Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this section 3 shall survive the termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than employees and on-site contractors, excluding Mentor Graphics' competitors, whose job performance requires access and who are under obligations of confidentiality. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation, the licensing and assignment provisions shall be binding upon your successors in interest and assigns. The provisions of this section 4 shall survive the termination or expiration of this Agreement.

5. **LIMITED WARRANTY.**

   5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED TO YOU FOR A LIMITED TERM OR LICENSED AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

   5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

6. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 6 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

7. **LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 7 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

8. **INDEMNIFICATION.** YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH YOUR USE OF SOFTWARE AS

DESCRIBED IN SECTION 7. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE EXPIRATION OR TERMINATION OF THIS AGREEMENT.

9. **INFRINGEMENT.**

9.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright or misappropriates a trade secret in the United States, Canada, Japan, or member state of the European Patent Office. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the infringement action. You understand and agree that as conditions to Mentor Graphics' obligations under this section you must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to defend or settle the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

9.2. If an infringement claim is made, Mentor Graphics may, at its option and expense: (a) replace or modify Software so that it becomes noninfringing; (b) procure for you the right to continue using Software; or (c) require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

9.3. Mentor Graphics has no liability to you if infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you make, use or sell; (f) any Beta Code contained in Software; (g) any Software provided by Mentor Graphics’ licensors who do not provide such indemnification to Mentor Graphics’ customers; or (h) infringement by you that is deemed willful. In the case of (h) you shall reimburse Mentor Graphics for its attorney fees and other costs related to the action upon a final judgment.

9.4. THIS SECTION IS SUBJECT TO SECTION 6 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

10. **TERM.** This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 1, 2, or 4. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics’ reasonable satisfaction.

11. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export any Software or direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

12. **RESTRICTED RIGHTS NOTICE.** Software was developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

13. **THIRD PARTY BENEFICIARY.** For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth herein.

14. **AUDIT RIGHTS.** You will monitor access to, location and use of Software. With reasonable prior notice and during your normal business hours, Mentor Graphics shall have the right to review your software monitoring system and reasonably relevant records to confirm your compliance with the terms of this Agreement, an addendum to this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet report log files that you shall capture and provide at Mentor Graphics’ request. Mentor Graphics shall treat as confidential information all of your information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement or addendum to this Agreement. The provisions of this section 14 shall survive the expiration or termination of this Agreement.

15. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** THIS AGREEMENT SHALL BE GOVERNED BY AND CONSTRUED UNDER THE LAWS OF THE STATE OF OREGON, USA, IF YOU ARE LOCATED IN NORTH OR SOUTH AMERICA, AND THE LAWS OF IRELAND IF YOU ARE LOCATED OUTSIDE OF NORTH OR SOUTH AMERICA. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section 15. This section shall not restrict Mentor Graphics' right to bring an action against you in the jurisdiction where your place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

16. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

17. **PAYMENT TERMS AND MISCELLANEOUS.** You will pay amounts invoiced, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Some Software may contain code distributed under a third party license agreement that may provide additional rights to you. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 060210, Part No. 227900